

DTIC FILE COPY

AD-A217 979



(21)

DTIC
FILED
JAN 24 1990
S D

Proceedings of the Seventh Annual National Conference on Ada Technology

March 13-16, 1989

Sponsored by
ANCOST, INC.

With Participation by
United States Army
United States Navy
United States Air Force
United States Marine Corps
FAA Technical Center

Co-Hosted by
Monmouth College
Penn State/Harrisburg
Jersey City State College
Cheyney University
Stockton State College

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

PROCEEDINGS OF SEVENTH ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

Sponsored By:
ANCOST, INC.
Washington DC.

Mike Saponter
405/355-9280
Le...

With Participation By:
United States Army
United States Navy
United States Air Force
United States Marine Corps
FAA Technical Center

Co-Hosted By:

Monmouth College
Jersey City State College

Cheyney University

Penn State/Harrisburg
Stockton State College

Bally's Park Place Hotel, Atlantic City, NJ
March 13-16, 1989

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Approved for Public Release: Distribution Unlimited

90 01 23 178

7th ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

CONFERENCE COMMITTEE 1988-1989

MR. ELMER F. GODWIN, Director
GEF Associates
Shrewsbury, NJ 07702

MR. JIM BARBOUR
Digital Equipment Corporation
Marrimack, NH 03054

MR. MIGUEL A. CARRIO, JR.
Teledyne Brown Engineering
Fairfax, VA 22030

DR. PHILIP CAVERLY
Jersey City State College
Jersey City, NJ 07305

MR. MICHAEL DANKO
General Electric Co
Moorestown, NJ 08057

DR. MARY R. ELLIS
Hampton University
Hampton, VA 23668

MR. DONALD C. FUHR
Tuskegee University
Tuskegee, AL 36088

MS. JUDITH M. GILES
Intermetrics, Inc.
Cambridge, MA 02138

MS. DEE M. GRAUMANN
General Dynamics, DSD
San Diego, CA 92138

DR. GEORGE C. HARRISON
Norfolk State University
Norfolk, VA 23504

DR. ARTHUR JONES
Morehouse College
Atlanta, GA 30314

DR. MURRAY KIRCH
Stockton State College
Pomona, NJ 08240

DR. GENEVIEVE M. KNIGHT
Coppin State College
Baltimore, MD 21216

DR. RICHARD KUNTZ
Monmouth College
W. Long Branch, NJ 07764

DR. RONALD LEACH
Howard University
Washington, DC 20059

MS. SUSAN MARKEL
TRW
Fairfax, VA 22031

MS. CATHERINE PEAVY
Martin Marietta Information
and Communication Systems
Denver, CO 80201-1250

DR. M. SUSAN RICHMAN
The Pennsylvania State Univ.
at Harrisburg
Middletown, PA 17057

MR. JOHN W. ROBERTS
EDO Corp.
Chesapeake, VA 23320

MS. CHARLENE ROBERTS-HAYDEN
GTE Government Systems
Needham Heights, MA 02194

MR. WALTER ROLLING
Ada Technology Group, Inc.
Washington, DC

MS. SUSAN ROSENBERG
Cadre Technologies, Inc.
Providence, RI 02903

MS. RUTH RUDOLPH
Computer Sciences Corporation
Moorestown, NJ 08057

MR. MICHAEL SAPENTER
Telos Federal Systems
Lawton, OK 73501

MR. JAMES SCHELL
Consultant
Ocean, NJ 07712

MR. TERENCE P. STARR
General Electric Company
Philadelphia, PA 19101

MR. CHARLES TANTILLO
Stockton State College
Pomona, NJ 08240

MR. JAMES E. WALKER
Network Solutions
Vienna, VA 22180

MR. JESSE WILLIAMS
Cheyney State University
Cheyney, PA 19319

ADVISORY MEMBERS

MR. BRIAN BAKER
Navy Department
Washington, DC 20350-2000

MAJOR GEORGE BEDAR
Marine Corps Tactical Systems
Support Activity
Camp Pendleton, CA 92055

MR. LOUIS J. BONA
FAA Technical Center
Atlantic City Airport, NJ 08405

CPT. SHEILA BRYANT
Marine Corps Tactical Systems
Support Activity
Camp Pendleton, CA 92055

MR. DANIEL E. HOCKING
AIRMICS
Atlanta, GA 30332-0800

MR. ALBERT RODRIGUEZ
HQ. CECOM
Ft. Monmouth, NJ 07703-5000

MAJOR DOUG SAMUELS
HQ AFSC/PLRT
Andrews AFB, DC 20334-5000

MS. KAY TREZZA
HQ CECOM
Ft. Monmouth, NJ 07703-5000

PANELS AND TECHNICAL SESSIONS

Tuesday, March 14, 1988

9:00 AM Welcoming Remarks
10:00 AM Para. 1
2:00 PM Session 1
2:00 PM Session 2
2:00 PM Session 3
2:00 PM Session 4

Ada Policy, Practices and Initiatives
Applications
Project Management
Distributed Processing
Performance Measurements

Wednesday, March 15, 1989

8:30 AM Panel II
10:45 AM Session 5 (Students' Presentations)
10:45 AM Session 6 (Students' Presentations)
10:45 AM Session 7 (Students' Presentations)
2:00 PM Session 8
2:00 PM Session 9
2:00 PM Session 10
2:00 PM Session 11
3:45 PM Session 12
3:45 PM Session 13
3:45 PM Session 14

STARS Technology Update
Ada & Education
Lessons Learned and Experimentation
Ada Development Issues
Applications
Project Management
Technology Research
Education/Training
Life Cycle Environments
Life Cycle Management
Reuse

Thursday, March 16, 1989

8:30 AM Panel III
8:30 AM Session 15
8:30 AM Session 16
2:00 PM Session 17
2:00 PM Session 18
2:00 PM Session 19
3:45 PM Panel IV

DoD Software Engineering Contractor's Capabilities Assessment
Reuse
Life Cycle Productivity
Testing and Evaluation
Designing for Ada
Life Cycle Productivity
Looking to the Future with Ada

PAPERS

The papers in this volume were printed directly from unedited reproducible copies prepared by the authors. Responsibility for contents rests upon the authors and not the symposium committee or its members. After the symposium, all the publication rights of each paper are reserved by their authors, and requests for republication of a paper should be addressed to the appropriate author. Abstracting is permitted, and it would be appreciated if the symposium is credited when abstracts or papers are republished. Requests for individual copies of papers should be addressed to the authors.

PROCEEDINGS

SEVENTH NATIONAL CONFERENCE ON ADA TECHNOLOGY

Bound—Available at Fort Monmouth

2nd Annual National Conference on Ada Technology Proceedings—1984—\$10.00
3rd Annual National Conference on Ada Technology Proceedings—1985—\$10.00
4th Annual National Conference on Ada Technology Proceedings—1986—\$15.00
5th Annual National Conference on Ada Technology Proceedings—1987—(Not Available)
6th Annual National Conference on Ada Technology Proceedings—1988—\$20.00
7th Annual National Conference on Ada Technology Proceedings—1989—\$25.00

Extra copies: 1-3 \$25.00; next 4-10 \$20; next 11 & above \$15.00 each

Make check or bank draft payable in U.S. dollars to the Annual National Conference on Ada Technology and forward request to:

Annual National Conference on Ada Technology
U.S. Army Communications-Electronics Command
ATTN: AMSEL-RD-SE-CRM (Ms. Kay Trezza)
Fort Monmouth, New Jersey 07703-5000

Telephone inquiries may be directed to Ms. Kay Trezza at (201) 532-1898

Photocopies—Available at Department of Commerce. Information on prices and shipping charges should be requested from:

U.S. Department of Commerce
National Technical Information Service
Springfield, Virginia 22151
USA

Include title, year, and AD Number

2nd Annual National Conference on Ada Technology—1984-AD A142403
3rd Annual National Conference on Ada Technology—1985-AD A164338
4th Annual National Conference on Ada Technology—1986-AD A167802
5th Annual National Conference on Ada Technology—1987-AD A178690
6th Annual National Conference on Ada Technology—1988-AD A190936

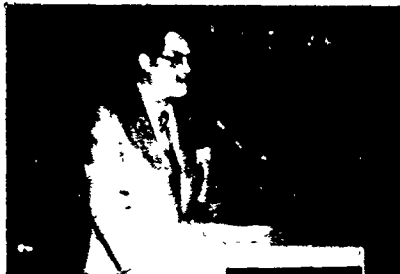
HIGHLIGHTS

Sixth Annual National Conference on Ada Technology

March 14-18, 1988

Crystal Gateway Marriott Hotel, Arlington, VA

Greetings



Mr James E Schell—Director, Emeritus, Center for Software Engineering, U.S. Army CECOM, Fort Monmouth, NJ



Dr. Jesse C Lewis—Vice President for Academic Affairs, Norfolk State University, Norfolk, VA



Dr. Joseph E. Gilmour—Executive Assistant to the Chancellor, University of Maryland, College Park, MD

Guest Speakers



MG Billy M Thomas—Commanding General, U.S. Army CECOM, Fort Monmouth, NJ



LTJG Emmett G. Paige, Jr., (Retired)—Commander, U.S. Army Information Systems Command, Fort Huachuca, AZ



Dr. Alan B. Salisbury, VP Contel Corp., Fairfax, VA



Mr. Norman Augustine—CEO, Martin Marietta Corp., Bethesda, MD



BG F. Russell Baldwin—Commanding General, Seventh Signal Brigade, Fort Ritchie, MD



COL Archie Taylor—PM, Common Hardware/Software, Fort Monmouth, NJ

Opening Session Panel Members



MG Eric Nelson—Deputy Commanding General, ESD, Hanscom Air Force Base, MA



MG Alonzo E. Short, Jr.—Deputy Commanding General, U.S. Army Information System Command, Fort Huachuca, AZ



RADM Harry S. Ouast—Director, Information Systems Division, Department of the Navy, Washington, DC



Mr. Loren Diedrichsen—Principal Technical Advisor, NATO Communications and Information Systems Agency (NACISA), Brussels



Ms. Virginia Castor—Director, Ada Joint Program Office, Arlington, VA



Dr. Larry Druffel—Director, Software Engineering Institute, Carnegie Mellon Institute, Pittsburgh, PA

Special Recognition Awards



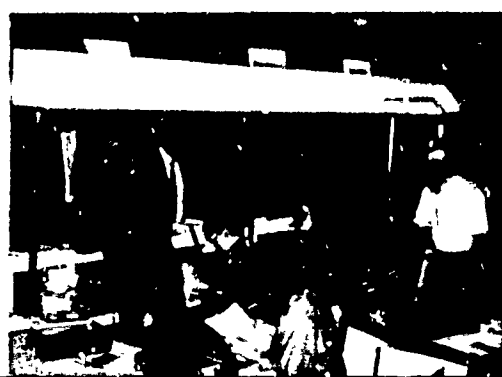
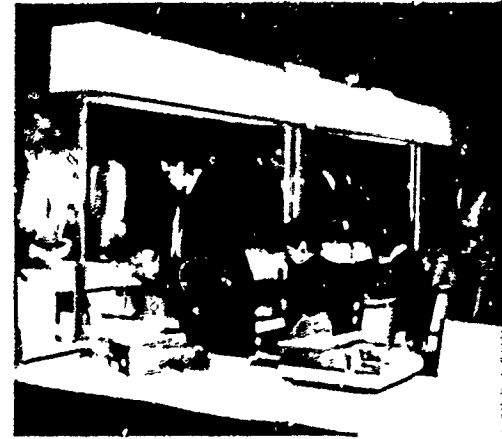
Mr. James E. Schell—Director, Emeritus, Center for Software Engineering, U.S. Army CECOM, Accepting Retirement Plaque for AN-COST Committee.



LTG Emmett Paige Jr.—(Retired), Commander, U.S. Army Information Systems Command, Fort Huachuca, AZ, accepting from Mr. Schell a Sword in Recognition of his Dedication to the Symposium and Pending Retirement from the Army.



Mr. James E. Schell—Accepting a Certificate of Recognition and Scholarship Award in His Honor, from Dr. Weldon Jackson, VP Academic Affairs, Moorehouse College, Atlanta, GA

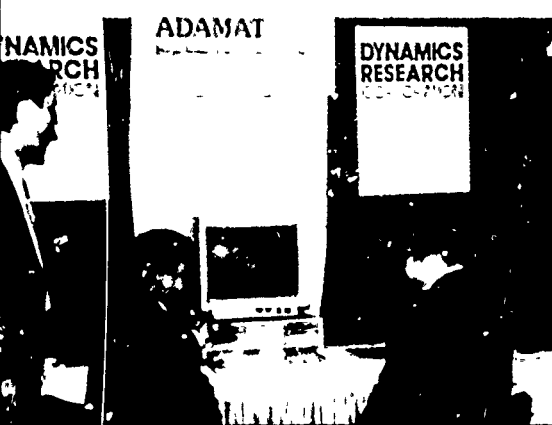
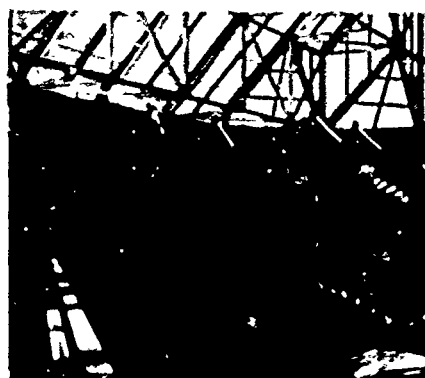




Admissions
and
Panel
Discussions



1988
Ada CONFERENCE
EXHIBITORS





SPEAKERS
Breakfast
730A

HIGHLIGHTS
OF THE
Ada TECHNOLOGY
CONFERENCE

**Ada
CONFERENCE**

PANEL DISCUSSION II
STARS TECHNOLOGY UPDATE

**SESSION
IN PROGRESS**



TABLE OF CONTENTS

TUESDAY, MARCH 14, 1989—8:00 AM-12:00 N

Grand Ballroom—(Blenheim and Marlborough Rooms)

OPENING RECEPTION:

Presiding:

Mr. John H. Sirtic, Acting Director Center for Software Engineering, U.S. Army Communications-Electronics Command, Fort Monmouth, NJ

Greetings:

Dr. William J. Maxwell, President, Jersey City State College, Jersey City, NJ
 Dr. LeVerne McCummings, President, Cheyney University, Cheyney, PA
 Dr. Ruth Leventhal, Provost, Penn State, Harrisburg Campus, Middletown, PA
 Dr. Richard Kuntz, Vice President, Monmouth College, West Long Branch, NJ
 Dr. Vera King Farris, President, Stockton State College, Pomona, NJ

PANEL DISCUSSION I Ada Policy, Practices and Initiatives (DoD Ada Executive Officials)

Moderator:

Mr. James E. Schell, President, SOPHSYS, Inc. 1

Panelists:

LTG Bruce R. Harris, Director of Information Systems for C⁴, Office of the Secretary of the Army, Washington, DC 1
 LTG Gordon E. Fornell, Commander, Electronic Systems Division, U.S. Air Force Systems Command, Hanscom Air Force Base, MA 2
 RADM Paul E. Tobin, Jr., Director of Navy Resources Management, Office of the Assistant Secretary of the Navy for Financial Management, Washington, DC 3
 Dr. R. E. Lyon (SES), Acting Associate Director for Engineering and Technology, Defense Communications Agency, Arlington, VA 3
 BG John D. Wakelin, Deputy Director, Unified and Specified Command C³ Support, Organization of the Joint Chiefs of Staff, Washington, DC 4

LUNCH, 12:00 N-2:00 PM

Ocean Ballroom (Rooms A & B)

Spotlight Speaker:

MG Billy M. Thomas, Commanding General, U.S. Army Communications-Electronics Command, Ft. Monmouth, NJ 5

Luncheon Speaker:

Mr. Joseph M. Del Balzo, Executive Director For System Development, FAA, Washington, DC 5

TUESDAY, MARCH 14, 1989—2:00 PM-5:30 PM

Grand Ballroom—Salons A, B, and C (Dennis)

SESSION 1: APPLICATIONS

Chairperson: Mr. James E. Walker, Network Solutions, Vienna, VA

RAMORA: Reusable Ada Modules for Optimal Resource Allocation—S. R. Mackey, Lockheed Austin Division, Austin, TX 9
 GENESYS: Embedded Software Tailorability—S. A. Bailey, J. D. Laird, and M. Angevine, Intermetrics Inc., Huntington Beach, CA 13
 An Ada Implementation of the Data Encryption Standard in a Real Time Environment—L. Grosberg and D. Coe, U.S. Army CECOM, Fort Monmouth, NJ 25
 A Hardware Independent System Development Approach Involving Ada—T. Dale, Unisys Defense Systems, McLean, VA 30

Implementation of Blackboard Systems in Ada—V. S. Dobbs and P. P. Cook, Wright State University, Dayton, OH 37

The AN/TSC-99 Redesign Effort—An Experience in Software Engineering with Ada—E. P. Gunderson, D. A. Vaughn, and G. E. Bostic, II, Telos Federal Systems, Shrewsbury, NJ 44

Grand Ballroom—Blenheim Room

SESSION 2: PROJECT MANAGEMENT

Chairperson: Charlene Roberts Haydon, GTE Government Systems, Needham Heights, MA

Practical Approach to Methodologies, Ada and DOD-STD-2167A—K. S. Ellison and W. J. Goulet, Jet Propulsion Laboratory, Pasadena, CA 51
 Lessons Learned in the Preparation of a Software Engineering Exercise—J. P. Fitzgibbon and G. Peavy, Martin Marietta Information and Communications Systems, Denver, CO 58
 A Comparison of Methods which Address the Development of Real-Time Embedded Systems—R. Guilloyle and R. Pirchner, Monmouth College, W. Long Branch, NJ; L. Von Gerichten, M. Ginsberg, and D. Clarson, Teledyne Brown Engineering, Inc., Eatontown, NJ 67
 Techniques for Optimizing Ada/Assembly Language Program Interfaces—E. N. Schacht, Computer Sciences Corporation, Huntsville, AL 78
 System Simulation in Ada for the Project Manager—K. J. Cogan, Electronics Technology and Devices Laboratory, Fort Monmouth, NJ; P. W. Caverly and C. Marino, Jersey City State College, Jersey City, NJ 87
 The Ada Software Development Methodology Evaluation and Selection Process: Fact or Myth?—S. J. McCullough, Computer Technology Group, Ltd., Washington, DC 93

Grand Ballroom—Salons A, B, & C (Marlborough)

SESSION 3: DISTRIBUTED PROCESSING

Chairperson: Ms. Dee Graumann, General Dynamics, DSD, San Diego, CA

An Ada Designed Distributed Operating System—M. B. Serkin, Martin B. Serkin & Co., Granada Hills, CA 100
 PARSIM: A Parallel and Real-Time Simulator for Concurrent Programs—M. D. Coleman and R. J. Leach, Howard University, Washington, DC 109
 Real-Time Pattern Recognition in Ada: On the Formulation of Neutral Net Recognizers by Ada Tasking of Massively Parallel Multicomputers—W. Arden, Telos Federal Systems, Shrewsbury, NJ 114
 TASKIT: An Ada Simulation Tool Kit Featuring Machine Independent Parallel Processing—M. Angel and P. Juozitis, General Dynamics, San Diego, CA 122
 Ada Run-Time Environment Considerations for Simulation—S. Shastri, Concurrent Computer Corp., Tinton Falls, NJ 128

Garden Rooms (Longwood, Imperial, Berkshire & Tivoli)

SESSION 4: PERFORMANCE MEASUREMENTS

Chairperson: Mr. Michael Sapenter, Telos Federal Systems, Lawton, OK

Benchmarking the Real-Time Performance of Dynamic Ada Processes—A. J. Lee, W. R. Macre, and D. K. Dol, Lockheed Electronics Co., Inc., Plainfield, NJ 132

Establish and Evaluate Ada Runtime Features of Interest for Real-Time Systems—S. Lefkowitz and H. Greene, IIT Research Institute, Lanham, MD; and M. Bender, U.S. Army CECOM, Fort Monmouth, NJ.....	139
Real-Time Performance Benchmarks for Ada—A. Goel, TAMSCO, Eatontown, NJ.....	145
Real-Time Ada Demonstration Project—M. E. Bender, U.S. Army CECOM, Fort Monmouth, NJ; and T. E. Grist, LabTek Corp., Woodbridge, CT.....	154
Modification of LU Factorization Algorithm for Parallel Processing Using Tasks Supported by Ada Language—S. N. Shah, Norfolk State University, Norfolk, VA.....	162

WEDNESDAY, BANQUET 7:00 PM-9:30 PM

Ocean Ballroom (Rooms A & B)

Guest Speaker: D. J. Herman, UNIX International

WEDNESDAY, MARCH 15, 1989—8:30 AM-10:30 AM

Grand Ballroom (Blenheim & Marlborough Rooms)

PANEL DISCUSSION II: STARS TECHNOLOGY UPDATE

Chairperson: COL Joseph S. Greene, Jr., U.S. Air Force, Director of STARS Program, Washington, DC

Panelists:

James King, System Architect, Boeing Aerospace, Kent, WA
J. W. Moore, System Architect, IBM, Galtersburg, MD
Terri Payton, System Architect, Unisys, Reston, VA
(Invited)

Grand Ballroom—Salons A, B & C (Dennis)

SESSION 5: STUDENT PRESENTATIONS—ADA & EDUCATION

<i>Chairperson:</i> Dr. Mary Ellis, Hampton University, Hampton, VA	
Learning Ada from Ada—L. Smithmier, Jr., University of Mississippi, University, MS.....	168
Problems in Using Ada as a Development Tool—A. J. Mull, University of Mississippi, University, MS.....	172
An Ada System for the Parallel Execution of FP Programs—N. Graham, Oklahoma State University, Stillwater, OK.....	176
Transferring from Pascal or C to Ada—J. Scholtz and S. Wiedenbeck, University of Nebraska, Lincoln, NE.....	182
Two Approaches to Ada: The Procedural (PASCAL) Approach and the Object Oriented (C+ +) Approach—G. R. Thompson, Morehouse College, Atlanta, GA.....	185

Grand Ballroom—Blenheim Room

SESSION 6: STUDENT PRESENTATIONS—LESSONS LEARNED AND EXPERIMENTATION

Chairperson: Dr. Philip Caverly, Jersey City State College, Jersey City, NJ

On Inclusion of the Private Part in Ada Package Specification—S. Muralidharan, The Ohio State University, Columbus, OH.....	18.
What Is the Object in Object Oriented Programming—K. V. Chan and W. Tsung-Juang, University of Mississippi, University, MS.....	193
A Two-Phase Reproduction Method for Ada Tasking Programs—M. M. Najjar and T. Elrad, Illinois Institute of Technology, Chicago, IL.....	197
Problems Encountered in Learning Object Oriented Design Using Ada—G. Carlson, St. Cloud State University, St. Cloud, MN.....	209
Quest for Usability in Ada Generics—K. Minder, Trenton, State College, Trenton, NJ.....	213

Grand Ballroom—Salons A, B & C (Marlborough)

SESSION 7: STUDENT PRESENTATIONS—ADA DEVELOPMENT ISSUES

Chairperson: Dr. Richard Kuntz, Monmouth College, W. Long Branch, NJ

Design Considerations Affecting Implementation of Byzantine Agreement Protocols in Ada—S. Hartman, Southeastern Massachusetts University, North Dartmouth, MA.....	218
Upgrading a Lisp Prototype (Advisor) to a System in Ada—M. Johnson, K. Robinson, and R. Washington, Hampton University, Hampton, VA.....	224
An Implementation of the Standard Math Functions in Ada—J. A. Frush, University of Mississippi, University, MS.....	226
Decomposition Schemes for Static and Dynamic Analysis of Ada Programs—R. Gopal, Vanderbilt University, Nashville, TN.....	230

LUNCH, 12:00 N-2:00 PM

Ocean Ballroom (Rooms A & B)

Spotlight Speaker:

BG John A. Hedrick, Commanding General, U.S. Army Information Systems Engineering Command, Ft. Huachuca, AZ.....

6

Luncheon Speaker:

LTG Jerry Max Bunyard, Deputy Commanding General, Research, Development and Acquisition, USAMC, Alexandria, VA.....

6

WEDNESDAY, MARCH 15, 1989—2:00 PM-3:30 PM

Grand Ballroom—Salons A, B & C (Dennis)

SESSION 8: APPLICATIONS

Chairperson: Mr. Brian Baker, Navy Department, Washington, DC

An Object-Oriented Approach to Simulating a Real-Time System in Ada—J. Margono and J. E. Walker, Network Solutions, Inc., Vienna, VA.....	239
Ada Implementation of Operating System Dependent Features—M. I. Schwartz and R. W. Hay, Martini Marietta Information & Communications Systems, Denver, CO.....	245
Implementation of a Real-Time Elevator Control Simulation System Using the Ada Language—D. Bagley, K. Land, H. Tamburro, and M. Vega, U.S. Army CECOM, Fort Monmouth, NJ.....	251

Grand Ballroom—Blenheim Room

SESSION 9: PROJECT MANAGEMENT

Chairperson: Ms. Catherine Peavy, Martin Marietta Information and Communication Systems, Denver, CO

Procurement of Air Traffic Control Software in Ada—A. C. Chung, FAA Technical Center, Atlantic City International Airport, NJ.....	257
The Use of a Software Engineering Exercise During Source Selection—D. G. Montgomery, The MITRE Corporation, FAA Technical Center, Atlantic City International Airport, NJ.....	262
An Approach to Ada Compiler Acceptance Testing—E. Amoroso and T. Nguyen, AT&T Bell Labs, Whippany, NJ.....	266

Garden Rooms (Longwood, Imperial, Berkshire & Tivoli)**SESSION 10: TECHNOLOGY RESEARCH****Chairperson:** Mr. Jesse Williams, Cheyney University, Cheyney, PA

Software Metrics Analysis of the Ada Repository— <i>R. J. Leach</i> , Howard University, Washington, DC.....	270
Ada Implementation of Sequential Correspondent Operations for Software Fault Tolerance— <i>P. N. Lee</i> and <i>A. Tamboli</i> , University of Houston, Houston, TX...	278
Ada—POSIX— <i>T. Fong</i> , U.S. Army Information Systems Engineering Command, Fort Huachuca, AZ.....	284

WEDNESDAY, MARCH 15, 1989—2:00 PM-5:00 PM**Grand Ballroom Salons A, B & C (Marlborough)****SESSION 11: EDUCATION/TRAINING****Chairperson:** Dr. Murray Kirch, Stockton State College, Pomona, NJ

Ada Summer Seminar—Teaching the Teachers— <i>M. S. Richman</i> , Penn State University of Harrisburg, Middletown, PA; <i>C. G. Petersen</i> , Mississippi State University, MS; and <i>D. C. Fuhr</i> , Tuskegee University, Tuskegee, AL.....	288
Training COBOL Programmers in Ada— <i>J. C. Agrawal</i> , Embry Riddle Aeronautical University, Daytona Beach, FL.....	295
Teaching Ada From the Outside-In— <i>D. P. Purdy</i> , Manatee Community College, Bradenton, FL.....	303
An Intermediate-Level Problem Set for Experienced Programmers or Writing Ada Code that Achieves the Language Goals— <i>R. S. Rudolph</i> , Computer Sciences Corp., Moorestown, NJ.....	307
A 10-Day Ada Course for the Industry— <i>F. Molnien</i> , Cameron University, Lawton, OK.....	313
Integrating Ada Training with Software Development— <i>P. Fortin</i> and <i>F. L. Moore</i> , Texas Instruments, Inc., Dallas, TX.....	316
The SEI Education Program— <i>H. Gibbs</i> , Carnegie-Mellon University, Pittsburgh, PA (Invited)	
Evaluation of Teaching Software Engineering Requirements Analysis (SERA)— <i>J. Sodhi</i> , Telos Federal Systems, Lawton, OK.....	321

WEDNESDAY, MARCH 15, 1989—3:45 PM-5:00 PM**Grand Ballroom Salons A, B & C (Dennis)****SESSION 12: LIFE CYCLE ENVIRONMENTS****Chairperson:** Mr. Albert Rodriguez, U.S. Army CECOM, Fort Monmouth, NJ

Ada Abstract Data Types—The Foundation of an Interactive Ada Command Environment— <i>J. A. Thalhamer</i> , <i>W. P. Loftus</i> , <i>C. L. Oel</i> , and <i>R. A. Foy</i> , Unisys Corp., Paoli, PA.....	326
A Software Engineering Documentation Environment— <i>T. J. Wheeler</i> , U.S. Army CECOM, Fort Monmouth, NJ.....	333
Documentation Generation System— <i>D. Rodericks</i> , <i>I. Rivera</i> , <i>B. Kolofsky</i> , <i>R. Quinones</i> , U.S. Army CECOM, Fort Monmouth, NJ.....	342
Reducing Software Development Costs with Ada— <i>J. R. Carter</i> , Martin Marietta Astronautics Group, Denver, CO.....	348

Grand Ballroom—Blenheim Room**SESSION 13: LIFE CYCLE MANAGEMENT****Chairperson:** Mr. Walter, Rolling Ada Technology Group, Inc., Washington, DC

The National Training Center Move and Upgrade: A Distributed Ada System— <i>D. Pottinger</i> , Science Applications International Corp., San Diego, CA.....	358
Software Quality Assurance in an Ada Environment— <i>S. Barkataki</i> , California State University, Northridge, CA; and <i>J. Kelly</i> , Jet Propulsion Laboratory, Pasadena, CA.....	362
Implementing Software First with Today's Technology— <i>E. J. Gallagher, Jr.</i> , U.S. Army CECOM, Fort Monmouth, NJ; <i>E. Fedchak</i> , IIT Research Institute, Rome, NY; and <i>D. Preston</i> , IIT Research Institute, Lanham, MD.....	368
Lessons Learned in Developing Requirements— <i>G. Healer</i> , Lockheed Engineering & Sciences Company, Houston, TX.....	383

Grand Ballroom—Salons A, B & C (Marlborough)**SESSION 14: REUSE****Chairperson:** Mr. Daniel Hocking, AIRMICS, Atlantic, GA

Tangram-L—A Program Description Language for Ada— <i>E. E. Doberkat</i> , University of Essen, West Germany.....	390
AdaL, An Automated Code Reuse System— <i>G. C. Harrison</i> , Norfolk State University, Norfolk, VA.....	404
Reusable Subsystems from a High Performance Ada Communication System— <i>T. L. Chen</i> and <i>W. Sobkiw</i> , E-Systems, Inc., St. Petersburg, FL.....	411

THURSDAY, MARCH 16, 1989—8:30 AM-12:00 N**Grand Ballroom (Blenheim Rooms)****PANEL DISCUSSION III: DoD SOFTWARE ENGINEERING CONTRACTOR'S CAPABILITIES ASSESSMENT****Chairperson:** Miguel A. Carrio, Jr., Teledyne Brown Engineering, Fairfax, VA**Panelists:**

Catherine H. Peavy, Martin Marietta Information & Communications Systems, Denver, CO
Hal Hart, TRW, Inc., Redondo Beach, CA
Barbara Nash, GTE Government Systems, Rockville, MD
Paul Mauro, Hughes Aircraft Co., Ground Systems Group, Fullerton, CA
 (Invited)

Grand Ballroom—Salons A, B & C (Marlborough)**SESSION 15: REUSE****Chairperson:** Ms. Ruth Rudolph, Computer Sciences Corp., Moorestown, NJ

Constructing Domain-Specific Ada Reuse Libraries— <i>J. Solderitsch</i> , <i>K. C. Wallnau</i> , and <i>J. A. Thalhamer</i> , Unisys Corp., Paoli, PA.....	419
Ada, Hypertext, and Reuse— <i>L. Latour</i> , University of Maine, Orono, ME.....	434
Disciplined Reusable Ada Programming for Real-Time Applications— <i>F. Arico</i> and <i>A. Gargaro</i> , Computer Sciences Corp., Moorestown, NJ.....	443
The Morehouse Object-Oriented Reuse Library System— <i>A. M. Jones</i> and <i>R. Bozeman</i> , Morehouse College, Atlanta, GA.....	456
Reuse and the Software Life Cycle— <i>D. S. Gulndi</i> , <i>W. M. McCracken</i> , and <i>S. Rugaber</i> , Georgia Institute of Technology, Atlanta, GA.....	463

Grand Ballroom—Salons A, B & C (Dennis)

SESSION 16: LIFE CYCLE PRODUCTIVITY

Chairperson: Mr. James Baibour, Digital Equipment Corporation, Merrimac, NH

- A Logical Framework for Version and Configuration Management of Ada Components—A. T. Jazaa and O. P. Brereton, University of Keele, Staffordshire, Great Britain..... 469
- Designing for Change: An Ada Design Tutorial—J. A. Hager, HRB-Systems, Inc., State College, PA..... 475
- A Portable Ada Implementation of Blocked_IO—J. J. Cupak, Jr., HRB-Systems, Inc., State College, PA..... 483
- Developing a Universal Ada Test Language for Generating Software/System Integration and Fault Isolation Test Programs—J. Ziegler, J. M. Grasso, L. Burgermeister, and L. D. Mollard, ITT Avionics, Nutley, NJ..... 494
- A DIANA Query Language for the Analysis of Ada Software—C. Byrnes, The MITRE Corp., Bedford, MA..... 511
- Ada Portability Among Heterogeneous—B. Casado and N. Bazzi, U.S. Army CECOM, Fort Monmouth, NJ.. 519

LUNCH, 12:00 N-2:00 PM

Ocean Ballroom—Rooms A & B

Spotlight Speaker:

MG Peter A. Kind, Program Executive Officer, Army Tactical Command and Control System (ATCCS), Fort Monmouth, NJ..... 7

Speaker:

Dr. Larry Druffel, Carnegie Mellon University, Director, Software Engineering Institute, Pittsburgh, PA..... 8

THURSDAY, MARCH 16, 1989—2:00 PM-3:30 PM

Grand Ballroom—Blenheim Rooms

SESSION 17: TESTING AND EVALUATION

Chairperson: CPT Sheila Bryant, Marine Corps Tactical Systems Support Activity, Camp Pendleton, CA

- Ada Compiler Validation: Purpose and Practice—R. Williams and P. Brashear, SolTech, Inc., Fairborn, OH; and S. Wilson, Wright-Patterson AFB, OH..... 522
- How to Live with TEXT_IO—D. W. Jones, Ridgecrest, CA..... 528
- Automatic Test Data Generation and Assertion Testing for Ada Program Units—L. Mayes, R. W. Aragon, D. Terrien, and J. Trost, Intermetrics, Inc., Huntington Beach, CA..... 537

Grand Ballroom—Salons A, B & C (Marlborough)

SESSION 18: DESIGNING FOR ADA

Chairperson: Dr. Ronald Leach, Howard University, Washington, DC

- Practical Advice for Designing Ada System Architectures—C. D. Buchman, Allied-Signal Aerospace Co., Teterboro, NJ..... 549
- Ada Design Tool—K. Tupper, M. Levitz, J. Helzron, S. Barlev, and P. Davarzo, Unisys S&GSG, Great Neck, NY..... 557
- Objects with Multiple Representations In Ada—K. M. George, Oklahoma State University, Stillwater, OK; and J. Sodhi, Telos Federal Systems, Lawton, OK..... 567

Grand Ballroom—Salons A, B & C (Dennis)

SESSION 19: LIFE CYCLE PRODUCTIVITY

Chairperson: Ms. Susan Markel, TRW, Fairfax, VA

- A Software Development Tool Using Ada—Pseudo Code Management System—D. Blau Liu, California State University—Long Beach, Long Beach, CA..... 576
- Data Reduction: An Ada Generics Methods—W. D. Ferguson, C. L. Carruthers, B. J. Carter, Jr., K. A. Staples, Jr., and C. F. Wise, General Electric Corp., ESD, Moorestown, NJ..... 584
- A Method of Translating Functional Requirements for Object-Oriented Design—R. Brown and V. Dobbs, Wright State University, Dayton, OH..... 589

THURSDAY, MARCH 16, 1989—3:45 PM-5:45 PM

Grand Ballroom (Blenheim Rooms)

PANEL DISCUSSION IV: LOOKING TO THE FUTURE WITH ADA

Chairperson: Miguel A. Carrio, Jr., Teledyne Brown Engineering, Fairfax, VA

Panelists:

- Jean Ichblah, Alsys, Inc., Waltham, MA
- Charles McKay, University of Houston-Clearlake, Houston, TX
- Frank Belz, TRW, Inc., Redondo Beach, CA
- Howard Yudkin, Software Productivity Consortium, Reston, VA
- (Invited)

OPENING PANEL

ADA POLICY, PRACTICES AND INITIATIVES



Mr. James E. Schell
President
SOPHSYS, Incorporated
(Moderator)

James E. Schell is the former Director of the CECOM Center for Software Engineering. He retired from the U.S. Civil Service as a Senior Executive (SES) in March 1988. He holds a Baccalaureate Degree in Mathematics, Physics, and French from Morehouse College, Atlanta, Georgia. He has studied graduate work in the MBA program and Anthropology at California State University, Northridge, California and completed several executive development courses at the University of California, Berkeley and the University of Southern California.

He has had an extensive and rewarding career in both government and industry. He has held training positions with the Air Force and Army Signal Corps; a Soldier in the Signal Corps; Signal Publications; a charter member of Command Control Information Systems in 1970 (CCIS-70) Project Manager's Office. From there, he held positions in Litton Data Systems Division as Director of the AN/TTC-39 Program Office and Director of the TACFIRE/TOS Program Office.

He returned to Government in 1979 in the Senior Executive Service as Director, U.S. Army Center for Tactical Computer Systems (CENTACS) during which time he founded this Ada Technology Conference. Mr. Schell culminated his government career by serving as the Deputy Program Manager, Army Command and Control System from 1985 to 1986 and as Director Center for Software Engineering from 1986 to 1988.

He now has his own consultancy as SOPHSYS, Inc. He and his wife, the former Doris Hunter, live in Ocean Township, New Jersey.



LTC Bruce R. Harris
Director of Information Systems for C4
Office of the Secretary of the Army
Washington, DC

Lieutenant General Bruce R. Harris was born in Sullivan County, Indiana on 13 August 1934. Upon completion of the Reserve Officers Training Corps curriculum and the educational course of study at Tennessee Technological University in 1956, he was commissioned a second lieutenant and awarded a Bachelor of Science degree in Business. He also holds a Master of Science degree in Political Science from Auburn University. His military education includes completion of the Signal Officer Basic and Advanced Courses, the United States Army Command and General Staff College, and the Air War College.

He has held a wide variety of command and staff positions culminating in his current assignment as Director of Information Systems for C4, Office Secretary of the Army. These include command of the 13th Signal Battalion, 1st Cavalry Division; command of the Division Support Command, 2d Armored Division; Chief of Staff and later Deputy Commandant of the U.S. Army Signal School; Deputy Assistant Secretary of Defense for Legislative Affairs; Assistant Division Commander, 9th Infantry Division; command of the U.S. Army Communications Electronics Engineering Installation Agency; Deputy Commander, U.S. Army Information Systems Command; and command of the U.S. Army Signal School.

Awards and decorations which General Harris has received include the Distinguished Service Medal, Legion of Merit, the Bronze Star Medal, the Meritorious Service Medal (with Oak Leaf Cluster), several Air Medals and the Army Commendation Medal. He also wears the Parachutist Badge and the Master Army Aviator Badge.

He is married to the former Claudia Alley and they have four children: Bruce, Jr., Mary Kathryn Brooks, Tim, and Brad.

OPENING PANEL

ADA POLICY, PRACTICES AND INITIATIVES



LTG Gordon E. Fornell
Commander Electronics Systems Division
U.S. Air Force Systems Command
Hanscom Air Force Base, MA

Lieutenant General Gordon E. Fornell is commander of Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

General Fornell was born on 18 September 1936, in Chicago and graduated from Maine Township High School, Des Plaines, Illinois, in 1954. He received a Bachelor of Science degree in Mechanical Engineering from Michigan State University in 1958 and a Master of Business Administration degree from the Wharton School, University of Pennsylvania, in 1978. While at Michigan State University, the general was a member of the 1957 Big Ten Conference championship swim team. He completed Squadron Officer School in 1963 and Air War College in 1973.

He was commissioned as a second lieutenant through the Air Force Reserve Officer Training Corps program and entered active duty in October 1958. He received his initial pilot training at Moore Air Base, Texas, and Greenville Air Force Base, Mississippi, from November 1958 to November 1959. In June 1960 he completed F-86L fighter-interceptor training at Perrin Air Force Base, Texas.

Upon graduation from Air War College in June 1973, the general was assigned to Headquarters, U.S. Air Force, Washington, D.C., as chief, Aeronautical Systems Division, Directorate of Development and Acquisition, until January 1977. In this capacity he was responsible for most of the aircraft development programs, including the F-15, B-1, F-16 and A-10, and for technology based programs involving aircraft equipment, life support and turbine engines.

General Fornell returned to the Pentagon in July 1981 as Deputy Director of Development and Production on the Air Staff and was responsible for programs that included aircraft, propulsion, avionics, armament and electronic combat systems.

In October 1982, General Fornell became special assistant for intercontinental ballistic missile modernization matters. Office of the Deputy Chief of Staff for Research, Development and Acquisition, at Air Force Headquarters. He became the Senior Military Assistant to the Secretary of Defense in January 1987. In that position he assisted and advised the secretary on the full range of defense responsibilities and national security matters. He assumed his present command in September 1988.

His military decorations and awards include the Defense Distinguished Service Medal, Distinguished Service Medal, Legion of Merit (with one Oak Leaf Cluster), Distinguished Flying Cross (with two Oak Leaf Clusters), Meritorious Service Medal (with one Oak Leaf Cluster), Air Medal (with 11 Oak Leaf Clusters), and Air Force Commendation Medal. He also wears the Basic Parachutist Badge and the Missile Crew Member Badge.

He was promoted to lieutenant general 1 October 1988, with same date of rank.

General Fornell is married to the former Barbara A. Bauer of LaGrange Park, Illinois. They have two children Kirsten and Eric.

OPENING PANEL

ADA POLICY, PRACTICES AND INITIATIVES



RADM Paul E. Tobin Jr.
Director of Navy Resources Management
Office of the Assistant Secretary
of the Navy for Financial Management
Washington, DC

Rear Admiral Paul E. Tobin, Jr. was graduated from the U.S. Naval Academy in 1963 and reported to the USS Tawara where he served as First Lieutenant and Main Propulsion Assistant. In 1968, Admiral Tobin was awarded a Masters of Science degree in Computer Systems Management from the Naval Post Graduate School. He was graduated with distinction by the Naval War College from the Naval Command and Staff Course. He was also graduated with distinction from the Industrial College of the Armed Forces in 1984.

Admiral Tobin has held a number of Command assignments among them the USS TATNALL in the Indian Ocean and the Persian Gulf; the USS FOX deployed to the Indian and the Western Pacific; he also assumed command of the Surface Warfare Officers School Command in July 1986. His awards and decorations include the Bronze Star, the Meritorious Service Medal and the Navy Commendation Medal (with two Gold Stars).

Admiral Tobin is currently assigned as Director, Department of the Navy Information Resources Management, he also serves as Director, Information Management Support Division under the cognizance of the Chief of Naval Operations.

Admiral Tobin is married to the former Lynne Carter of Shaker Heights, Ohio. They have two daughters: Mary Elizabeth and Patricia.



Dr. R.K. Lyon (SES)
Acting Associate Director for
Engineering and Technology,
Defense Communications Agency
Arlington, VA

Dr. Lyon received the Bachelor of Science Degree in Electrical Engineering from MIT in 1950. His graduate degrees include the MS and PhD degrees in Electrical Engineering from the University of Maryland, and the professional Electrical Engineers degree from MIT.

Dr. Lyon is Acting Associate Director for Engineering and Technology, Defense Communications Agency. Among his other duties, he is the DCA Ada Executive. His office is the interface between DCA and the world of high technology in government, industry and academia, and has the responsibility to promote the transition of high technology to operational use in the CJ community. Formerly Special Assistant to the Director, Information Processing Techniques Office, DARPA, and Deputy Director, Defense Communications Engineering Center, DCA, Dr. Lyon has been involved in systems engineering of U.S. CJ systems since 1970. He has 39 years of professional experience in DoD information and communications systems and has previously held position with the IBM Corporation and National Security Agency.

He is a senior member of the IEEE, a Governor of the International Council for Computer Communications, a member of the Armed Forces Communications and Electronics Association, and a recipient of the Defense Communications Agency Director's Exceptional Civilian Service and Distinguished Executive Awards.

OPENING PANEL

ADA POLICY, PRACTICES AND INITIATIVES



MC John B. Wakelin
Deputy Director, Unified and Specified
Command C³ Support
Organization of the Joint Chiefs of Staff
Washington, DC

Brigadier General John B. Wakelin was born in San Francisco, California on 12 January 1936. Upon completion of the Reserve Officers Training Corps curriculum and the educational course of study at the University of San Francisco in 1959 he was commissioned a second lieutenant and awarded a Bachelor of Science degree in Philosophy. His military education includes completion of the Basic Signal Officer Course, the Infantry Advanced Officer Course, the United States Army Command and General Staff College, and the National War College.

He has held a wide variety of important command and staff positions culminating in his current assignment. Immediately prior he served as Deputy Commander for Research and Development at the Communications Electronics Command, Fort Monmouth, New Jersey. Other key assignments held recently include Commander of the 35th Signal Brigade at Fort Bragg, North Carolina and Special Assistant to the Commanding General of the Communications Electronics Command, Fort Monmouth, New Jersey.

General Wakelin has an extensive background in the management of communications and signal resources. Following overseas service as Chief of Communications Electronics with the United States Defense Liaison Group in Indonesia, he commanded the 50th Signal Battalion (Airborne) at Fort Bragg, North Carolina. General Wakelin then served as Deputy Commander for the battalion's parent 35th Signal Group. Following completion at the National War College, he served as Deputy Director for Command, Control and Communications with the Defense Communications Agency, Washington, DC.

Awards and decorations which General Wakelin has received include the Bronze Star Medal, the Meritorious Service Medal (with Oak Leaf Cluster), and the Joint Service Commendation Medal. He also holds the Army Commendation Medal (with Oak Leaf Cluster) and is authorized to wear the Parachutist Badge.

He and his wife Janis (Jan) have five children: John, Jennifer, Jeffrey, Heather, and Jaclyn.

GUEST SPEAKERS

ADA POLICY, PRACTICES AND INITIATIVES



MC Billy M. Thomas
Command General
U.S. Army Communications-Electronics Command
Fort Monmouth, NJ

Major General Billy M. Thomas was born in Crystal City, Texas on 14 August 1940. He grew up in Killeen, Texas. Upon completion of the Reserve Officer's Training Corps curriculum and the educational course of study at Texas Christian University in 1962, he was commissioned a second lieutenant in the U.S. Army and was awarded a Bachelor of Science degree in Secondary Education. General Thomas holds a Master of Science degree in Telecommunications Operations from George Washington University.

His military education includes completion of the Signal Officer Basic and Advanced Courses, the Army Command and General Staff College, and the Army War College.

In addition to General Thomas' many important command assignments in Germany, Thailand, and Vietnam, he has also held a variety of significant staff assignments prior to his assuming the position of Commanding General, U.S. Army Communications-Electronics Command, among them were: Special Assistant to the Dean, National Defense University; Deputy Commanding General, U.S. Army Signal Center and School; Army Staff as the Deputy Director; Combat Support Systems, Office of the Deputy Chief of Staff for Research, Development and Acquisition.

Awards and decorations which General Thomas has received include the Legion of Merit, Bronze Star Medal, the Meritorious Service Medal and the Army Commendation Medal. He is also authorized to wear the Parachutist's Badge.

He is married to the former Judith K. McConnell of Boise, Idaho. They have four children: Jon, Kim, Kirsten, and David.



Mr. Joseph M. Del Balzo
Executive Director for System
Development, FAA
Washington, DC

As FAA's Executive Director for System Development, Joseph M. Del Balzo is one of four Executive Directors who assist the FAA Administrator in setting agency policy and governing the development and accomplishment of agency programs.

Mr. Del Balzo is responsible for National Airspace System Development, Advanced Design and Management Control, Airport System Development, and FAA's Technical Center in Pomona, New Jersey.

Mr. Del Balzo was Director of FAA's Eastern Region from November 1981 to July 1987. As Director of the Eastern Region, Mr. Del Balzo was responsible for all FAA activities within seven states (New York, New Jersey, Pennsylvania, Delaware, Maryland, Virginia, and West Virginia) and the District of Columbia.

Mr. Del Balzo was Director of the FAA Technical Center from January 1979 to November 1981. He previously served as the Technical Center Deputy Director. He was Chief of the Microwave Landing System (MLS) Division of the Systems Research and Development Service. He was responsible for the successful development and selection of the U.S. design of the MLS.

Mr. Del Balzo began his Federal career in 1958 as an Electrical Engineer in Portland Maine. He holds a B.S. in Electrical Engineering from Manhattan College, and an M.S. in Engineering Management from Drexel University. In 1975, on a one-year fellowship to Princeton University, he studied public policy and international affairs. He was awarded an Honorary Doctoral degree in Aeronautical Science by Embry-Riddle Aeronautical University in August 1981. He is a general aviation enthusiast and is an instrumented, multi-engine private pilot.

CONTEST SPEAKERS

ADA POLICY, PRACTICES AND INITIATIVES



Brig John A. Hedrick
Commanding General, U.S. Army
Information System Engineering Command
Fort Meachum, AZ

Brigadier General John A. Hedrick was born in Houston, Texas on 24 January 1941. After completion of the Reserve Officers Training Corps curriculum, he was commissioned a second lieutenant on 6 November 1964 and awarded the Bachelor of Science degree in Electrical Engineering at Texas A&M University. He also holds a Masters of Business Administration degree in Operations Research and Systems Analysis from Tulane University. His military education includes completion of the Signal Officer Basic and the Armor Officer Courses, the Radio Officer Course, the U.S. Army Command and General Staff College, and the U.S. Air Force War College.

He has held a wide variety of important command and staff positions culminating in his present assignment. Following his assignment as Operations Officer in the 9th Signal Battalion, he served as communications Staff Officer, National Communications System; Chief, Congressional Inquiry Division; Office of the Chief of Legislative Liaison; Deputy Commander, 1st Signal Brigade, Army Communications Agency Korea; Training and Doctrine System Manager, Army Tactical Communications System; Deputy Director for Plans, Programs and Systems; Office of the Director of Information Systems for C4; Office of the Secretary of the Army, Washington, DC.

His military decorations and awards include the Legion of Merit and the Bronze Star (each with two Oak Leaf Clusters), the Defense Meritorious Service Medal and the Army Meritorious Service Medal (with three Oak Leaf Clusters), and Army Commendation Medal, and the Army General Staff Identification Badge.

General Hedrick is married to the former Katherine Crain of Childress, Texas; they have three daughters: Janell, Janet and Jo Anne.



LTC Jerry Max Muryard
Deputy Commanding General
Research Development and Acquisition
USAMC, Alexandria, VA

Lieutenant General Jerry Max Muryard was born in Albus, Oklahoma on 2 April 1931. Upon completion of the Reserve Officers training Corps Course curriculum and the educational course of study in 1954, he was commissioned a second lieutenant and awarded a Bachelor of Science degree in Animal Husbandry. He also holds the Master of Science degree in International Relations from George Washington University. His military education includes completion of the Infantry Officer Basic course, the Field Artillery Officer Advanced Course, the U.S. Army Command and General Staff College, and the National War College.

He has held a variety of important command and staff positions culminating in his current assignment as Deputy Commanding General, Research, Development, and Acquisition; U.S. Army Materiel Command. Other key assignments held recently include Assistant Deputy Chief of Staff for Research, Development, and Acquisition; Project Manager for the Tactical Fire Direction System/Field Artillery Tactical Data Systems; Deputy Director for Defense Test and Engineering; Project Manager, PATRIOT Air Defense Missile Systems; and Commanding General U.S. Army Missile Command and Redstone Arsenal, Alabama.

Among his command assignments are the 2d Battalion, 20th Artillery, 1st Cavalry Division, Vietnam, Chief, Technical Support; U.S. Army Operational Test and Evaluation Agency, and Commander, Yuma Proving Ground, Arizona.

His awards and decorations include the Defense Superior Service Medal, The Legion of Merit, the Distinguished Flying Cross (with Oak Leaf Cluster), the Bronze Star Medal (with two Oak Leaf Clusters), the Meritorious Service Medal (with two Oak Leaf Clusters), several Air Medals, the Joint Service Commendation Medal, the Master Army Aviator Badge, the Office of the Secretary of Defense Identification Badge.

He is married to the former Celia Wilkerson; they have two children: Mike and Brenda.

CUEST SPEAKERS

ADA POLICY, PRACTICES AND INITIATIVES



MC Peter A. Kind
Program Executive Officer,
Army Tactical Command and Control
Systems (ATCCS)
Fort Monmouth, NJ

Major General Peter A. Kind is a native of Wisconsin. Upon completion of studies at the University of Wisconsin in 1961, he was commissioned a Second Lieutenant and awarded a Bachelor of Science degree in Economics. He also holds a Master of Business Administration from Harvard University. His military education includes the Basic Officer Course at the Signal School, the Communications Officer Course offered at the U.S. Marine Corps Amphibious Warfare School, the U.S. Army Command and General Staff College and the U.S. Army War College.

He was assigned to the 97th Signal Battalion (Army), 10th Special Forces Group (Airborne) in Germany and as Signal Advisor to the 21st Infantry Division (Air Assault) in Vietnam.

Following duty as Assistant Division Signal Officer of the 81st Airborne Division and as Executive Officer and S2/S3 (Intelligence/Operations and Training) for the 82d Signal Battalion, Fort Bragg, North Carolina, he served in the War Plans Division of the Office of the Deputy Chief of Staff for Operations and Plans, Headquarters, Department of the Army. He commanded the 1st Cavalry Division's 13th Signal Battalion, Fort Hood, Texas and studied at the Logistics Management Center's School of Management Science. General Kind then served as Chief of the Concepts and Studies Division, Directorate of Combat Developments at the Signal Center prior to Army War College attendance.

He then served as Commander of the 1st Signal Brigade with concurrent duty as the Assistant Chief of Staff, J6, U.S. Forces in Korea and G-6, Eighth U.S. Army; as Director of Combat Development, and as Deputy Commanding General

and Assistant Commandant, U.S. Army Signal Center and School, Fort Gordon, Georgia. He served as Deputy Controller of the NATO Integrated Communications System Central Operating Authority, NATO's equivalent to the U.S. Defense Communications Systems, headquartered at SHAPE, Belgium, prior to his appointment as Program Executive Officer, Command and Control Systems.

General Kind has been awarded the Legion of Merit, the Bronze Star Medal (with two Oak Leaf Clusters), and the Meritorious Service Medal (with two Oak Leaf Clusters). He is also the recipient of the Air Medal with two device and the Army Commendation Medal, the Senior Parachutist Badge and the Army General Staff Identification Badge.

He is married to the former Sandra L. Hanson. They have a son, Lee.

GUEST SPEAKERS

ADA POLICY, PRACTICES AND INITIATIVES



Dr. Larry Druffel
Carnegie Mellon University
Director, Software Engineering Institute
Pittsburgh, PA

Larry E. Druffel is Director of the Software Engineering Institute. Appointed to that position in September 1986, he was previously Vice President for business development at National, a company that provides advanced software development technologies.

Druffel has been associated with the Ada program since 1978. He was a member of the High Order Language Working Group, and became the first Director of the Ada Joint Program Office. He was later appointed Director of Computer Systems and Software in the Office of the Secretary of Defense (Research and Advanced Technology), a position that included management responsibility for the Ada program. He defined computer technology research strategies, was the initial Architect of the STARS program, and proposed the Software Engineering Institute.

Druffel, who was Associate Professor and Deputy Director of the Department of Computer Science at the United States Air Force Academy, has managed research programs in advanced software technology, artificial intelligence, and CII at the Defense Advanced Research Projects Agency.

In addition to being coauthor of a computer science textbook and of more than 30 professional papers, Druffel is also a software/Ada editor for Defense Science and Electronics. He holds a Bachelor's degree in Electrical Engineering from the University of London, and a Doctorate in Computer Science from Vanderbilt University. He is a senior member of IEEE and a member of the Association for Computing Machinery.



Donald J. Herman
Organizing Chairman
UNIX International

Mr. Donald J. Herman has been the Organizing Chairman of UNIX International since November 1985. Prior to this appointment, he served for 10 years with NCR Corporation of Dayton, Ohio and most recently as the Executive Vice President in the Office of the Chief Executive with responsibility for the company's integrated business units.

He holds a Bachelor's degree in Industrial Engineering from the University of South Dakota; he served as a Naval Officer from 1954 to 1960 in the National Security Agency.

In 1962, he was a founder of COMRESS, a highly successful software development and leasing firm. During his tenure as its Chairman and Chief Executive Officer, COMRESS founded four other computer related companies, one of which was CONTENT, a computer communications company; Mr. Herman assumed active management of CONTENT which subsequently acquired COMRESS.

In 1979, NCR acquired CONTENT and appointed Mr. Herman as the President and Chief Executive Officer of the independent operating Subsidiary, NCR CONTENT. Mr. Herman was later appointed a Vice President of NCR Corporation and went on to be named Chairman, NCR CONTENT, and was appointed Executive Vice President of NCR Corporation.

In 1985, while at NCR, he became the founding Chairman for the Corporation for Open Systems (COS). COS was one of the computer industry's first open systems consortia, and was primarily concerned with establishing the Open System Reference Model as a industry standard.

Don Herman's experience with high technology companies spans more than 26 years.

RAMORA : REUSABLE Ada MODULES FOR OPTIMAL RESOURCE ALLOCATION

Stephen R. Mackey

Lockheed Missiles & Space Company
Austin, Texas

Today, the tactical commander is confronted with a high technology battlefield that has significantly increased in complexity, speed, and diversity. Automated decision aids can drastically reduce the time required for decision-making processes necessary for planning combat operations. One key problem associated with such decision aids is developing methods to employ costly and scarce combat resources efficiently and effectively. Upgrading current tactical command, control, communications, and intelligence (C3I) systems or the fielding of new C3I systems warrants the application of this technology. By creating Ada-based prototype decision aid technology, development time and costs for emerging C3I systems can be significantly reduced. The RAMORA software is a decision aid consisting of reusable components that optimizes the allocation of specific resources (weapons) against selected requirements (targets) for a tactical scenario.

Overview

The Reusable Ada Modules for Optimal Resource Allocation (RAMORA) system was developed under contract to the Naval Research Laboratory (NRL) as part of the Software Technology for Adaptable, Reliable Systems (STARS) Planning and Optimization Algorithms Foundation Areas. The purpose of STARS was to create well designed and documented Ada software for reuse by agencies within the Department of Defense (DoD) as well as defense contractors.

RAMORA, a Computer Software Configuration Item (CSCI), is based on an object-oriented design emphasizing reusability and portability. At the beginning of software development, a reusable component search was performed on several government and non-government repositories to locate candidate software for reuse. Under the contract, three major reusable Computer Software Components (CSCs) were created within the RAMORA software: Effectiveness, Criteria, and Optimization (see Figure 1). This paper presents an overview of the RAMORA software components followed by a detailed description of the Effectiveness CSC, Criteria CSC, and the Optimization CSC.

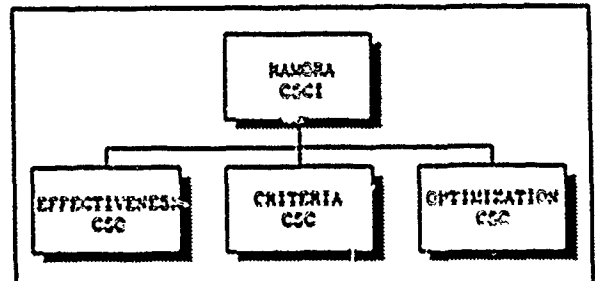


Figure 1. RAMORA Reusable CSCs

RAMORA Components

RAMORA is a decision aid which automates the currently manual technique of determining the most appropriate weapon systems for pairing against designated targets. A menu-driven interface allows for easy manipulation of input and constraint data. The following processes are facilitated in the RAMORA software:

- Development of a target list
- Development of a weapon systems list
- Determination of the weighting factor priorities
- Manual inclusion or exclusion of weapon/target pairs
- Optimization of target/weapon pairings
- Displaying the optimal solution

Target List

The user designates targets for weapon allocation by selecting from a generic-type target data base. A list of designated targets is developed in order to allocate the available weapon systems. Options for developing the current target list are add, delete, modify, and display (generic or current) targets. For each target in the current target list, the priority, probability of damage, kill type, dimensions, and latitude/longitude position are entered.

Weapon List

RAMORA tracks both an available and selected weapon data base. The user selects weapons from the

available list to be considered for assignment to designated targets. RAMORA allows for development of a current weapon systems list and associated data. Options for developing the weapon list are add, delete, modify, and display (available and current) weapon systems. For each weapon system in the selected weapon list, the platform, munition, forced deployment, platform range and speed, cost factor, scarcity factor, release conditions, delivery accuracies, status, and latitude/longitude position are entered.

Weighting Factors

Various factors are weighted into the weapon/target pairing calculations such as attrition, cost, damage, and scarcity of the weapon systems. RAMORA includes an interface to allow the user to set the weighting factors.

Manual Pairing

RAMORA allows the user to constrain weapon/target pairs through manual exclusions and inclusions of weapon/target pairs. An example of manual inclusion is when a weapons officer has orders to pair a particular weapon system against a designated target or set of targets. When solving for the solution, the optimization algorithm will make the forced pairing despite the cost or other contradictory factors.

Optimization

Once the target and weapon list, weighting factors, and possible manual inclusions/exclusions are entered, a measure of merit matrix is computed that represents the "cost" or criteria of allocating each weapon system to each designated target. The criteria calculations are based on weapon/target effectiveness, platform survivability, weapon scarcity, weapon cost, time on target, and the user-specified weighting factors. Weapon effectiveness estimates are defined in terms of single sortie probability of damage (SSPD) and the number of sorties necessary to damage a target to a prescribed level (NRQMT) for each weapon type. The optimization algorithm, using integer programming techniques, then assigns weapons to targets by minimizing the sum of the associated criteria values. The assignment is constrained by the available number of weapons of each type and the requirements for damage (NRQMT values).

Display of Results

After the optimization process, the solution matrix can be displayed for review and modifications. The optimal pairing of available weapons to targets is displayed in an easy-to-read, color-coded matrix. Multiple solution matrices with corresponding weapon/target lists can be saved and then restored for review.

Effectiveness CSC

The Effectiveness CSC is a reusable component responsible for manipulating the weapon effectiveness estimates. The Joint Munitions

Effectiveness Manual (JMEM) methodology was used to calculate each weapon effectiveness estimate, single-sortie probability of damage (SSPD), which is used to compute a cumulative damage probability, and the number of sorties required to damage a target to a specified level (NRQMT). The JMEM methodologies were developed and are maintained by the Joint Technical Coordinating Group for Munitions Effectiveness (JTCCG/ME) under the Joint Chiefs of Staff to model mathematically various weapon systems and subsystems for weapon effectiveness estimate computations. Two JMEM methodologies are incorporated within the RAMORA software, Air-to-Surface (JMEM/AS) and Surface-to-Surface (JMEM/SS), including an adaptation to the JMEM/AS methodology implementing helicopter deliveries.

Air-to-Surface

The JMEM/AS methodologies are open-end methods based on Gaussian delivery error and ballistic error distributions for a rectangular target area element, assuming the target elements are uniformly distributed throughout the target area. The damage functions are limited to representations that are integrable in closed form with respect to the Gaussian delivery error and ballistic error distributions. The JMEM/AS methodology considerations begin at munition release from the fixed wing airborne platform; therefore, flight profiles are not a dependent factor in the weapon effectiveness estimates. The JMEM/AS Basic Manual describes the derivation of the logic and mathematical expressions for the methodologies. The methodologies are modeled by five computer methods:

- Unguided Weapon against Non-runway, Unitary or Area type Targets
- Guided Weapon against Non-runway, Unitary or Area type Targets
- Projectile/Rocket against Non-runway, Unitary or Area type Targets
- Unguided Weapon against Runway type Targets
- Guided Weapon against Runway type Targets.

In addition to the five computer methods is the trajectory methodology that recomputes a more accurate and representative value for some of the release conditions than what is available in the tables for various effectiveness estimate computations.

The JMEM/AS computer methods use data from ten files implemented by the Automated Weaponing Optimization Program (AWOP/JMEM). The files are divided into the following dependencies:

- Weapon system dependency files - platform, munition, characteristics; platform and munition pairings; delivery accuracies and errors; and release conditions.
- Target dependency files - target characteristics and interdicted aircraft.
- Weapon/target dependency files - effectiveness indices and hard-target reliabilities.

Surface-to-Surface

The JMEM/SS methodology implemented in the RAMORA software originates from a manual for evaluating the effectiveness of nonnuclear indirect-fire weapons against area targets. The methodology is independent of any delivery aspects of the land platform when the weapon effectiveness estimates are computed. The JMEM/SS methodology is capable of calculating weapon effectiveness estimates for High-Explosive (HE) and approved Conventional Munition (ICM) type weapons.

A data format was developed for the RAMORA software based on AMOP/JMEM data files. Five files were developed and are represented in the following two groups:

- Weapon system dependency files - platform and munition characteristics, platform and munition pairings, and error probabilities
- Weapon/target dependency file - lethal areas.

The target data was obtained from the target file in the AMOP data base; therefore no additional target files were needed.

Helicopter Adaptation

While a specific JMEM methodology does not exist for helicopter delivery for air-to-surface munitions, the JMEM/AS methodology can be implemented to compute weapon effectiveness estimates. Because the munitions types, release condition parameters, and delivery accuracies for helicopter platforms are similar to that of fixed wing aircraft, the Unguided, Guided, and Projectile/Rocket JMEM/AS methodologies are used.

In calculating the weapon effectiveness estimates for weapon systems with helicopter platforms, the AMOP/JMEM file format used for the JMEM/AS calculations can be used. In addition to the ten AMOP/JMEM files, one file was developed to incorporate present and possible future parameters in delivering weapons from a helicopter platform.

Criteria CSC

The Criteria CSC is a reusable component responsible for manipulating the criteria values for the weapon/target pairs. The optimization algorithms evaluate the criteria values for pairing available weapon systems to designated targets. The criteria value calculations are sensitive to the baseline values, including the user-related weighting factors, and the priority ratio.

Baseline Values

The baseline values are factors relating to the characteristics of available weapon systems and targets. RAMORA uses baseline values sensitive to the effectiveness of the weapon system against the target, expected platform attrition rate to the target, scarcity of the weapon system, the unit cost

of the weapon system, and the time for the weapon system to reach the target. These baseline values are normalized, multiplied by the weighting factors, and, where appropriate, multiplied by the number of weapons required, NREQT. (For operational reasons, only one weapon type is paired to a given target at a specified time.) Each of these preliminary criteria values for each weapon/target pair is normalized by the maximum criteria value for all weapon/target pairs and then summed with the target's priority.

Priority Ratio

The priority ratio is a measure of the importance of priority when determining a solution to the weapon/target pairing problem. The priority ratio is used when resources are insufficient to meet the damage requirements of all the targets, and represents the criteria value assigned to a target when no weapon is allocated. The weapon/target pairing solutions will differ depending on whether the user desires to maximize the number of targets destroyed or to destroy higher priority targets at the expense of destroying more, lower priority targets. The priority ratio represents the number of next lower priority targets that would be equivalent to the higher priority target.

Optimization CSC

The Optimization CSC is a reusable component responsible for optimizing available weapon systems against designated targets. Optimization is based on the weapon/target effectiveness estimates, weapon/target criteria values, the target's priority, and the priority ratio. Two algorithms are implemented in RAMORA, Branch and Bound Search Technique and the Transportation Cases Method.

Branch and Bound

The branch and bound search technique is similar to an enumeration procedure (computing all possibilities), except that usually most of the non-optimal possibilities are discarded without being tested. The discarding occurs when an initial partial assignment is already too poor a selection to become optimal regardless of which remaining assignments are made.

The branch and bound search algorithm consists of two primary components, branching and bounding. The branching and bounding is similar to a search of a tree structure. The algorithm branches on the assignment of one of the remaining weapons and then bounds the best answer that could be expected after this assignment is made. This is an efficient solution to the weapon/target pairing problem due to the constraint of allocating only one weapon type to each designated target.

Transportation Cases Method

Another approach to solving the weapon/target pairing problem relies on the traditional

"transportation" problem. Each target has an associated demand for weapons, and weapons have an associated supply. The problem is to assign the weapons to the targets in a manner that satisfies the demand with the available supply while minimizing the sum of the associated criteria values. The problem with this model of the weapon/target pairing requirements is that the demand is actually a function of the weapon type and is a matrix of values, not one value per target.

A modification of the traditional transportation approach was developed by solving a series of transportation problems with each case consisting of groups of weapons and targets that can all be hit by the weapons. This method works well when most of the weapon classes consist of enough weapons to destroy all or most of the targets, or when each weapon class contains only enough weapons to destroy one or two targets at a time. For other cases, the computer run time may become prohibitively large for even relatively small (10 weapons, 10 targets) problems. A result of this algorithm is software for the generic transportation problem.

Combination of Algorithms

A complex scheme was implemented for the RAMORA software that incorporates both the multiple transportation cases method approach and the branch and bound approach. Each of these approaches works best where the other fails. Before optimizing, the NRCMT values, criteria values, and controlling parameters (size of problem, priority ratio, etc.) are evaluated to determine the number of transportation cases to be solved if that approach were taken. If the number of cases is less than some predetermined threshold, RAMORA uses the transportation approach to optimize; if not, RAMORA uses the branch and bound technique. In this manner, the true mathematical optimum is achieved in a timely way (see Figure 2).

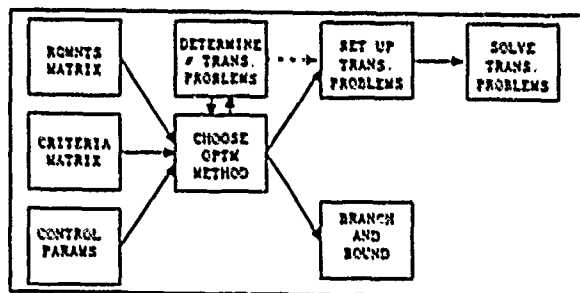


Figure 2. Optimization Algorithm Selection.

Conclusion

RAMORA is a decision aid automating manual techniques for optimizing allocated resources against designated requirements. RAMORA is based on using an object-oriented design emphasizing reusability and portability. Using object-oriented design, three reusable Computer Software Components were developed: Effectiveness, Criteria, and Optimization.

Reference

1. Joint Technical Coordinating Group for Munitions Effectiveness (Air-to-Surface) Methodology Working Group. Derivation of Joint Munitions Effectiveness Manual/Air-to-Surface Open-End Methods (61 JTCG/ME-3-7), May 1980.



Stephen R. Mackey
Lockheed Austin Division
6800 Burleson Road
T420/30F
Austin, Tx 78744

After graduating from the University of Texas at Austin with a Bachelor of Science in Mechanical Engineering, began work with Lockheed Austin Division within the Tactical CSI Independent Research and Development (IR&D) organization emphasizing in Operations Research. While working on various contracts and IR&D projects, have designed, developed, and implemented software relating to weapon effectiveness methodologies, aircraft survivability models, and other tactical CSI models.

GENESYS: EMBEDDED SOFTWARE TAILORABILITY

Stephen Bailey, James Laird, Gary Falacara, Mark Angwine

Intermetrics, Inc.
Huntington Beach, California

KEYWORDS: Ada Run-Time Support Environment (ARTSE),
Object-Oriented Programming,
Software Tailorability,
Software Reuse,
Software Development Tools.

A recurring predicament encountered in developing real-time embedded Ada software is the inflexibility of compiler Ada Run-Time Support Environments (ARTSE) to meet specific application requirements. That is, compilers lack support for tailoring and extending the ARTSE. The second short-coming of compiler ARTSE's is their lack of extendibility dictated by their dedication to run-time support for only Ada language constructs and semantics. In response to the need for a tailorable, extendable run-time support environment for Ada embedded applications, Aerojet ElectroSystems Corporation (AESC), Intermetrics, Inc. and Sparta Inc. have teamed to develop the GENeralized Embedded SYstem Specification (GENESYS) tool under the STARS Foundation program.

CONCEPTUAL BACKGROUND

A principle objective of the Ada language is to offer a high order medium for the development of reliable, portable software targeted to Embedded Computer Systems (ECS). A number of features are provided within the semantics of the Ada language to address this objective that traditionally have been considered the domain of Operating Systems (OS). Ada language primitives offer features such as multi-tasking, synchronization, time-based delays, interrupt handling and others without forcing the applications programmer to go outside of Ada language constructs. To achieve this, the Ada compiler vendor must provide the necessary control over the underlying machine to support the full semantic richness and functionality embodied within the Ada language. This support (ARTSE) is provided via a combination of compiler-generated code and a collection of external routines that map the abstractions of Ada onto the bare machine (typical of ECS's) or onto underlying real-time OS's. The ARTSE is not directly callable by the applications programmer. Rather, the applications code is resolved to ARTSE interfaces by the Ada Compilation System (ACS) according to the specific Ada language constructs employed by the applications programmer. For example, an Ada

application using tasking will have included into its executable load image the ARTSE code and routines necessary to support the multi-tasking paradigm on the hardware to which the ACS is targeted. The intent of such a scheme is, in part, to enhance the portability of the applications code by providing the applications programmer a means to access powerful features without directly interfacing to the machine or OS. A major weakness of this approach however is the inability of the applications developer to modify or "tailor" the critical components that may impact the run-time behavior of their application. An ECS often requires high degrees of efficiency that may not be attainable with the standard ARTSE components and algorithms. Obviously, no Ada compiler vendor can anticipate in advance all of the unique requirements of a given embedded system. Hardware configurations vary widely even where the instruction set architecture remains constant. Specialized I/O, mass storage, memory management, and co-processing schemes often characterize an embedded system architecture. For this reason, run-time source code that comprises the ARTSE is supplied with many vendors' embedded target ACS's for an additional licensing fee. This permits the modification and recompilation of the ARTSE. Using this approach, an application developer can incorporate specific drivers, interrupt handlers, and other critical run-time algorithms into the embedded systems' support software in place of generalized vendor supplied versions. In examining the approaches undertaken by various development projects and research efforts to tailor Ada run time behavior, we discovered two distinct strategies. In the first case, the source code of an ARTSE is directly modified without violating the semantic integrity of the Ada language that it is supporting. The second approach is to bypass the predefined capabilities of Ada that are deemed unsuitable via the use of applications level interfaces and services (user-supplied). Clearly any solution addressing the problem of run time tailoring and extension must support both approaches to achieve wide application and utility.

GENESYS provides a framework methodology and supporting tool to facilitate both the customization and reuse processes. GENESYS provides an automated and orderly way in which a user can easily manage and exploit tailorable and reusable components. GENESYS will not automatically alter an ARTSE component but it will strictly control and manage a set of altered

components - both ARTSE and applications level- and automatically tailor the construction of the complete executable program. GENESYS accomplishes this by assisting the user in the characterization of the system to be constructed and then by drawing upon an existing set of available tailored ARTSE alternative components in strict accordance with the user's specified requirements. GENESYS also provides a tool framework and supporting methodology for the growth and maintenance of a dynamic set of tailored and tailorable components. These components can be both ARTSE and/or non-ARTSE (e.g. applications level). In this sense, GENESYS can be characterized as a productivity enhancement tool and methodology that facilitates both program construction and general software reuse at both the ARTSE and the application layer.

GENESYS employs a set of abstractions that build upon and complement the standard architectural features of Ada to provide more powerful support for the construction and utilization of reusable software building blocks. Key to this strategy is the notion of discrete CLASSES of services, alternative implementations of these services or INSTANCES, the complete INHERITANCE tree of discrete components which comprise any given instance, and the set of quantifiable ATTRIBUTES which describe and characterize the behavior of any given INSTANCE of a CLASS.

Borrowing from object oriented programming techniques [STRO88], GENESYS introduces the notion

of a CLASS to build upon the Ada specification construct. In GENESYS a CLASS specifies a related set of visible exportable services (operations) and/or structures which are represented at the highest level by a single Ada package or subprogram specification. The lowest level Class is a single Ada specification implemented by one or more body variations. Higher level classes can be constructed which are aggregates of other Classes of lower level services and structures. Supporting and implementing this set of high level services are any number of underlying Ada bodies, specifications for "withed" units (CLASSES themselves) and their associated bodies. Within a CLASS there can exist any number of variant bodies and subclasses (withed subprogram units) that implement the given class (Ada specification). Thus, a GENESYS CLASS is the set of callable or visible operations and structures contained within an Ada specification as well as the entire "Inheritance" tree of all possible Ada subprogram bodies and supporting subclasses that implement the top level or "root" class. Since CLASSES can be constructed as aggregates of any number of lower level CLASSES, very complex tree-like structures can result that depict the potential component inheritance structure of the CLASS. Any Ada library specification can serve as the root of a class and newer more abstract CLASSES can always be built on top of combinations of lower level CLASSES. GENESYS manages the context necessary to support the building of these more abstract objects on top of combinations of lower level objects

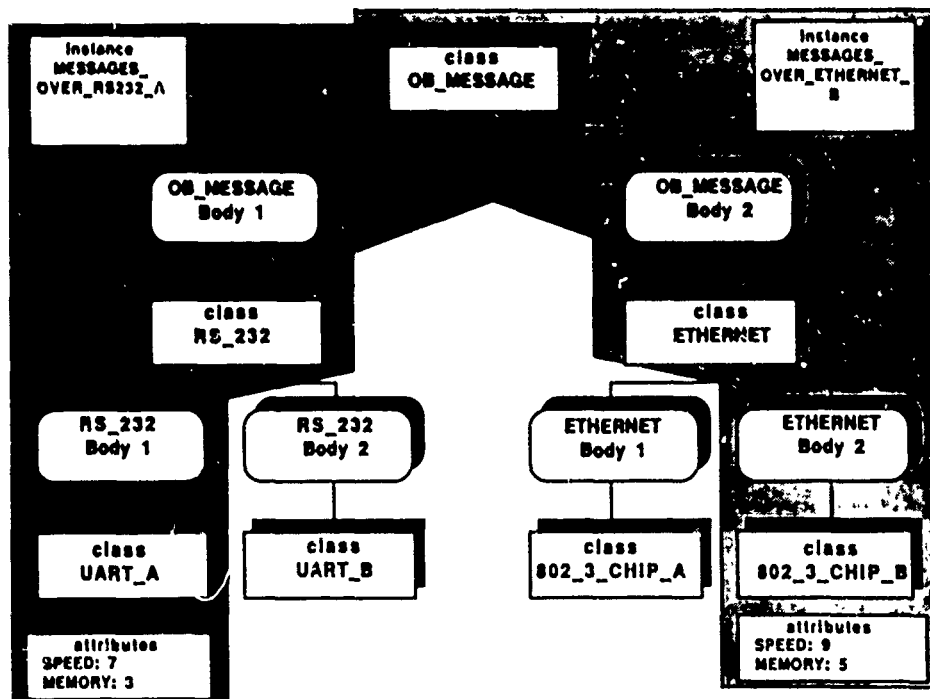


FIGURE 1. GENESYS BRINGS CLASSES & INSTANCES TO Ada

ensuring that the inheritance structure (the Ada dependency tree) and associated behaviors are maintained. In doing so, GENESYS promotes good object oriented programming practice by facilitating the reuse of well designed CLASSES for inclusion in other yet-to-be-developed higher level CLASSES.

Construction of a semantically correct executable program obviously requires the selection of a single Ada body for each specification. An INSTANCE is the path within the tree where choices have been made at each point that a body alternative (and its inheritance of withed units) is available. Figure 1 represents a high level CLASS of services -- On Board Message Passing-- with two very distinct implementations. Instance MESSAGES_OVER_RS232_A is comprised of those lower level units which implement the message passing services for a hardware target which communicates over an RS-232 bus while instance MESSAGES_OVER_ETHERNET_B implements the same set of high level message services for an ethernet based architecture. The notion of an INSTANCE definition and its associated inheritance structure of components insulates the user (a programmer) of such a service from the details of its implementation. It also provides a convenient method of managing differing versions or releases of software services. The user discriminates between instances based upon behavioral and other relevant attributes defined for a given CLASS and the variable empirical values assigned to those attributes for each separate INSTANCE that has been defined. The attribute characterizes the CLASS while the attribute values characterize each

INSTANCE of that CLASS. Referring to Figure 1 again we see that the two instances of the Message Passing service vary widely with respect to both speed and memory utilization thereby allowing a user to discriminate between them and make a selection based upon these two important behavioral attributes (assuming the communication hardware has not yet been selected).

Key to the GENESYS concept is the notion of a SESSION. The GENESYS SESSION contains all of the CLASS INSTANCE selections that a user has made in order to support the construction of a compiled and/or executable (linked) system. Once created, the SESSIONS are managed by GENESYS and can be saved, recalled, copied, modified, or deleted from the system.

The following sections discuss the installation of classes and instances, how a programmer would use GENESYS to select class instances and build a session, and how GENESYS fits into the software life cycle.

THE LIBRARIAN'S ROLE: CREATING & MAINTAINING CLASSES AND INSTANCES

In order to use GENESYS as a tool to assist in tailoring and reusing Ada components, it is necessary to first place those components, and a variety of information about them, in the GENESYS database. This is the librarian's role -- to collect components and place them in an organized fashion into a library where users will later be able to find them easily.

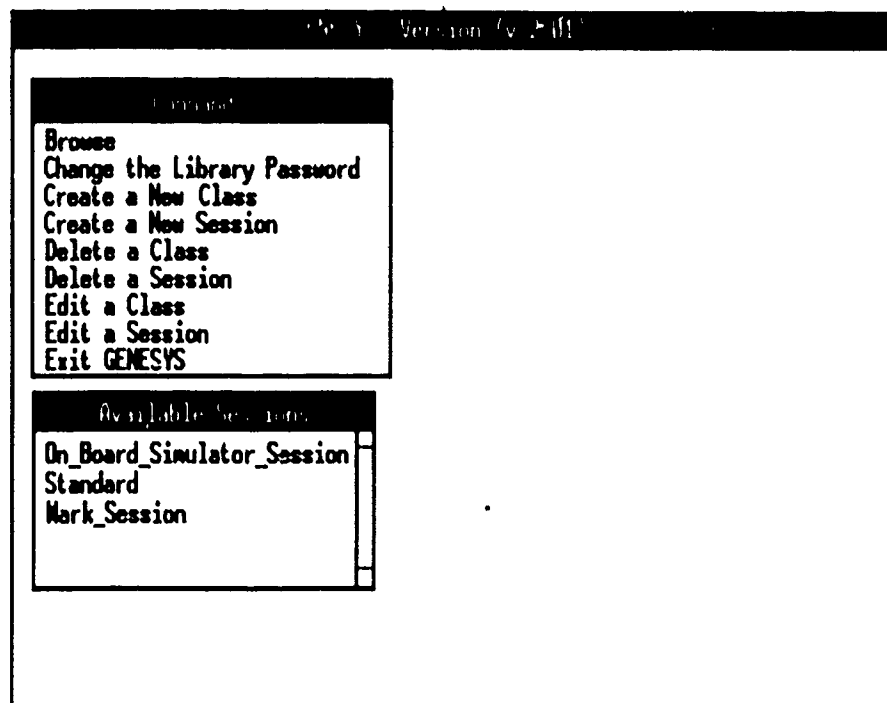


FIGURE 2. THE GENESYS INTERFACE IS EASY TO UNDERSTAND

GENESYS provides a simple, menu-driven user interface to assist in the librarian's duties. The user interface is based on the X windowing system, a standard package of routines for manipulating windows, icons, mouse and pointer (WIMP). Figure 2 illustrates the opening screen of GENESYS. In the upper left of the window is a "selectable list" of commands that can be invoked from this window. Clicking the mouse while the cursor is on one of these commands will cause the command to be invoked. If the first command in the list (BROWSE) is selected, two new elements appear on the screen: a message box (in the lower right) and a new selectable list (in the lower left) (see figure 3). The message box instructs the user to make a selection in one of the lists, and to press the "CONFIRM" button to cause the selection to take effect. This sequence is typical of the dialog between the user and GENESYS for all command functions. The use of the confirmation via the message box, plus the general avoidance of commands that require the user to type a response is designed to minimize user input errors or ambiguities concerning acceptable inputs.

For the librarian, there are four basic operations that need to be performed in GENESYS:

- o Adding new components (Ada specifications and bodies) to the database,
- o Creating new instances from existing components,
- o Deleting instances, and

- o Deleting Ada components from the database.

Each of these operations is accompanied by a series of dialogs between GENESYS and the user with GENESYS restricting the actions the user can take to ensure that inconsistent or erroneous installations cannot take place.

To install a new class in GENESYS, the user selects "Create a New Class" from the command list on the main screen, shown in figure 2, and then fills in the necessary information in response to queries from GENESYS. In particular, the user must provide:

- o The name of the new class,
- o The location of the source code file for the class' Ada specification,
- o The location of a help file describing the class,
- o The names of the attributes that will be used to distinguish the various instances of the class, and locations of attribute help files that describe the meaning of the attributes.

Next, the user must describe the relationships between the new class and existing components in the database. One simple rule dictates the order of installation of new components: Before any new specification or body can be added, all package specifications that are "withed" by the new

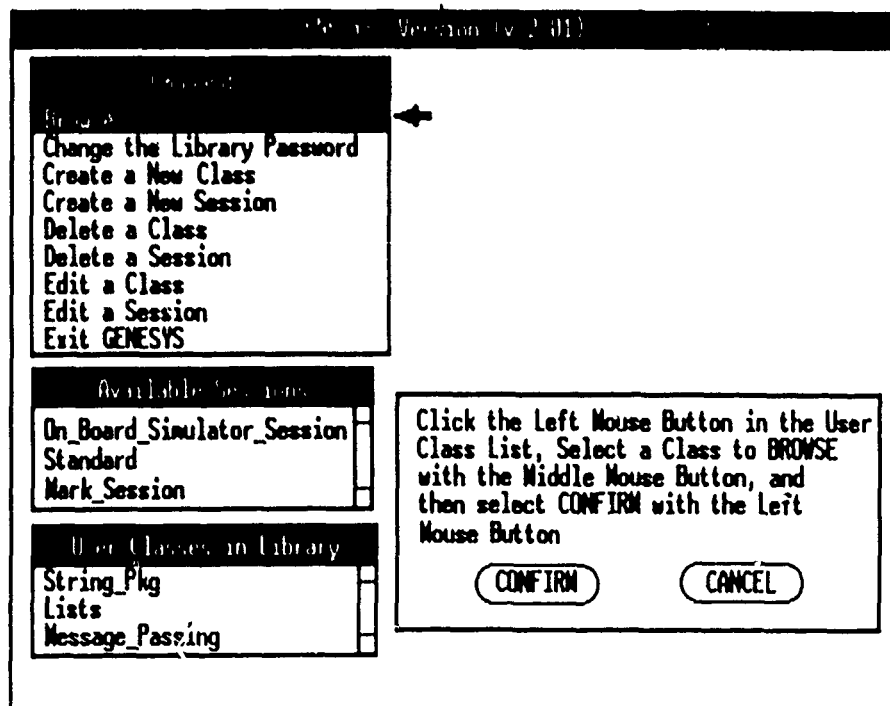


FIGURE 3. GENESYS GUIDES USERS THROUGH OPERATIONS

component must already be installed in the database. In other words, installation proceeds from the bottom-up. Low-level packages must be present before higher-level components that are directly dependent on the lower level components can be added. This is necessary to avoid any situation where an Ada specification or body in the database will not compile because it lacks supporting units. Likewise, when a class is to be removed from the database, it must be at the top of its "dependency tree" -- it cannot be removed if it is "withed" by any other component. GENESYS assures this component closure property at all times maintaining a consistent database.

The final step in the dialog for adding a new class is to indicate the connections to lower level components. GENESYS asks whether there are any "subclasses" to the installed classes. These "subclasses" are those package specifications that are "withed" by the class being installed. The librarian must enter the names of all subclasses that the class directly "withs." GENESYS checks to ensure that the subclasses have already been installed. This establishes the links within the database that will be used when components are retrieved, ensuring that all supporting components are also retrieved for compilation into the program library.

After a collection of components are placed in the database, it is necessary to create instances from the classes of these components. To define an instance, the librarian first chooses a class in the database. GENESYS then asks for some general information about the new instance to be defined:

- o The name of the instance,
- o Location of a help file describing the instance, and
- o The values of the attributes for the instance.

Recall that attributes for the class in general were defined when the class was installed. The attribute values are in the form of a numeric range (1..10). These values are used to assist the user in selecting a specific instance of the class that best fits their needs.

The librarian next proceeds down a "dependency tree," and must make a selection of a single body to support each specification encountered in the tree. GENESYS leads the user through the tree and lists the available alternative bodies at each branch in the tree. When this selection process is complete, the instance definition is entered into the database and now represents a complete, coherent suite of compilable Ada units able to provide the services defined by the class.

For deletion of components from the database, certain rules must be enforced by GENESYS to

protect the coherency of the database. Sessions can be deleted at any time. However, an instance can only be deleted if it is not included in any existing session. Deleting an instance currently in use would leave at least one session in an inconsistent state. Removing sessions and instances does not cause the removal of any actual Ada components from the library. To delete bodies and specifications, further prerequisites must be met.

To delete a class, it is necessary to reverse the process of adding components, and to delete from the "top down." Before a class can be deleted, any instances rooted in the class must first be deleted. If the user is able to delete all such instances safely, the class itself may be deleted. When this is done GENESYS automatically deletes the class specification and any set of available bodies in the database (since they are meaningless without the specification).

Individual body components can be removed at any time, but only if the body is not the last body present in the database to support its class specification. If there are multiple alternate bodies, then removals are permitted. However, removal of the last body would leave the corresponding specification part in an unusable, and therefore erroneous condition, and consequently is not permitted. The specification and the last body must be removed together.

By using this dialog framework for the librarian's basic chores of entering and deleting body and specification components and defining and deleting instances, the consistency and coherency of the GENESYS component library can be safely maintained. Another technique to assure safe database modification is to minimize the number of accesses to the database. A group of related additions or deletions is held by GENESYS until a complete set of modifications is accumulated. Then the entire batch of changes is sent via SQL to the database, thereby decreasing the chances of an interruption in the process (by system failure or other causes).

BUILDING APPLICATIONS WITH GENESYS

In the grand scheme of reuse (figure 4), GENESYS provides library management and application build capabilities. It permits a user to piece together an application by concentrating at the level of abstraction expressed by the class specifications he or she wishes to incorporate into the design. In other words, GENESYS's library management scheme encourages a building block approach to software development, where the user sees only the jagged edge of the block to which his application is intended to connect and the build capabilities insulate the user from the actual structure during the process of source code compilation. Thus the class specification literally specifies the tip of the iceberg to which the user wishes to attach

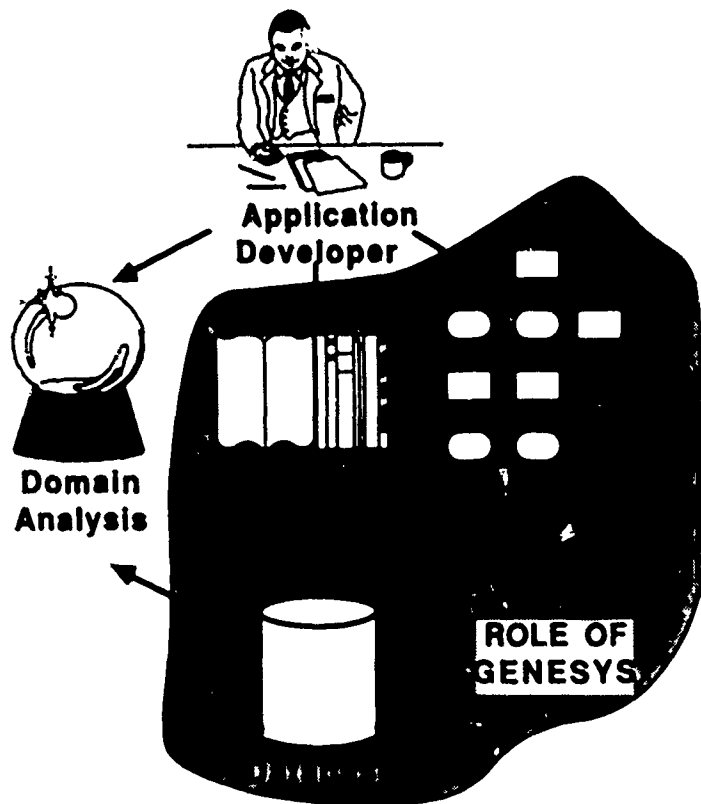


FIGURE 4. THE ROLE OF GENESYS IN REUSE.

(figure 5). Since such class specifications represent the embodiment of the reusable portion of an object and no more, GENESYS strongly embraces the software engineering principle of information hiding and encourages the use of abstract data types in designing classes.

There are six distinct steps in building an application through a GENESYS session. The following paragraphs detail the steps and provide an example of a GENESYS session.

Step 1: Identify an Application and Select a Component to Reuse. The process of using GENESYS is a multi-step process that first involves the selection of a class specification to use in building the target application. While GENESYS provides a capability for browsing through the component library, it is limited in aiding the user in making this selection (i.e. it has no true domain analysis capability). This is because GENESYS does not taxonomically classify its components or provide a user tailorable knowledge base to help a user decide which classes to use. There is no inherent reason why such capabilities could not be added to the tool, and indeed, as future funding and time permit, these important capabilities will be added.

Step 2: State User Constraints and Preferences. Initially, the restrictive vision of class

composition may seem somewhat uncomfortable (especially given the iceberg analogy). To alleviate this, GENESYS provides the notion of class attributes which can be used to express the concerns or constraints of the user regarding the iceberg he or she has just selected. Is it too large? Is it fast enough? Is it hardware specific or is it portable? It is this process of constraining the choice that becomes the second step in the process of using GENESYS to build an application.

Step 3: Select an Implementation. The next step is for the user to choose an implementation of that class that best fits within his application constraints. Attributes permit the developer to perform this second step without resorting to a primitive and time consuming source code analysis. GENESYS, using a scoring technique similar to Intermetrics' Reusable Software Library [BUNT87], makes recommendations based upon preferences stated by the user in terms of attribute values or permits the more knowledgeable user to select an instance by directly viewing a list from the library.

Step 4: Save the Selection(s). Once the selection is made, GENESYS performs the fourth step which is to record the user's selected classes and instances into a group known as a session, where the session represents the entire collection of class/instance pairs that the user has chosen for his application.

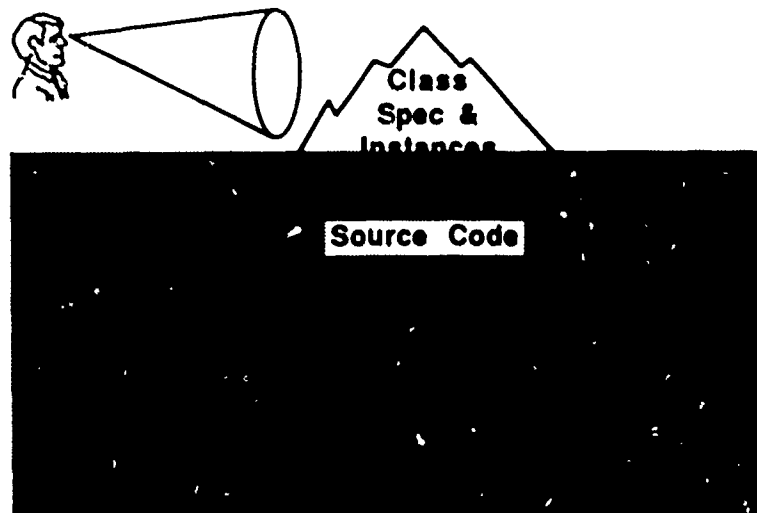


FIGURE 5. USER VIEW OF REUSABLE COMPONENT THROUGH GENESYS.

The session is versatile in that the user can create one or more sessions for each application, thereby keeping a stored variety of implementations for a single application that can be called up and built without having to rewrite application code. This capability permits a user to create an application and test it with several different versions of a reusable component in order to evaluate the performance.

Step 5: Generate a Representation of the Selections. Having a session, the user can then perform the fifth step of creating a Vendor Interface File (VIF), which is a textual representation of all the source code modules known to the library that are needed to build the application according to the user's specifications. The VIF is automatically created by GENESYS by analyzing the session class-instance pairs and information stored about the instances within the GENESYS database. Thus, in this fifth step, the user's abstract specification is turned into a script by which a compiler can build the application. In the initial version of GENESYS, we attempted to incorporate the user's code directly into the VIF but soon realized that the process of specifying a large user application could be quite tedious. For this reason, the current tool only addresses the source code modules needed to construct the classes selected from the GENESYS database. The VIF can be used, however, as a compilation order script to compile the source code into the users working Ada library or as an Ada Run Time Support Environment (ARTSE) reconstruction script for a Custom Build Tool (CBT) to use to alter the content of an ARTSE.

Step 6: Place the Selected Components Into the User's Library. This step in building an ARTSE or compiling code into a local library is the sixth and final step in using GENESYS and due to a special tool called the File Transfer Tool (FTT)

can be performed in a development environment where the GENESYS host system (a Sun workstation running X) and the target environment are separate but connected (by a network or RS-232 link).

Let us now consider an example using GENESYS to select alternative message passing algorithms.

Step 1: Identify an Application (i.e. Create a Session) and Choose a Component to Reuse (i.e. Select a Class). Let us consider a case in which we wish to employ a simple string message passing scheme in an application that will reside on distributed hardware. First, the user creates a session which will hold the specification for all of the components that will be used from the GENESYS library. This session is named `Message_Test`. The class that is selected is named `Message` and basically provides a simple scheme in which each task has a single queue to contain its messages. These queues can be addressed by using a unique `Task_ID` that must be assigned manually (and carefully) while designing the application(s). Class `Message` is represented by the following specification:

```
package MESSAGE is
  MAXIMUM_MESSAGE_LENGTH : constant POSITIVE :=
    4_096;

  subtype MESSAGE_TYPE is
    STRING(1..MAXIMUM_MESSAGE_LENGTH);

  type TASK_ID_TYPE is new INTEGER range 1..255;

  procedure RECEIVE(
    RECEIVER : in TASK_ID_TYPE;
    SENDER   : out TASK_ID_TYPE;
    MESSAGE  : out MESSAGE_TYPE);

  procedure SEND(
    DESTINATION : in TASK_ID_TYPE;
```



```

SOURCE      : in TASK_ID_TYPE;
MESSAGE     : in MESSAGE_TYPE;

```

end MESSAGE;

Step 2: State User Constraints and Preferences (i.e. Set Attribute Values). Class Message has three simple attributes, namely Speed, Memory and Reliability where a high Speed value reflects a fast message passing instance, a high Memory value reflects the usage of a large amount of memory by the instance and a high Reliability value reflects an instance in which messages have a high probability of arriving at their destination correctly. The user of our hypothetical class expresses the following preferences (on a scale of 1..10):

```

Speed      = 9 (High Speed)
Memory     = 5 (Moderate Memory
               Utilization)
Reliability = 2 (Low Reliability Tolerated)

```

Step 3: Select an Implementation (i.e. Ask for an Instance Recommendation). Assuming that class Message has several instances as shown in figure 6, GENESYS recommends the Fast_Buffered_Messages instance over the Packetized_Messages_With_Retries and Unbuffered_Messages instances. This instance is recommended by the tool because of its speed and moderate memory utilization.

Step 4: Save the Selection(s) (i.e. Save the Session). The user then instructs GENESYS to accept the recommendation and save it in the session Message_Test. This information is updated both in memory, where the user works, and in the database from which it can be later retrieved. At this point the session itself corresponds to the application being developed and contains the specification of a class (Message) and an implementation of that class (Fast_Buffered_Messages) needed by the application.

Step 5: Generate a Representation of the Selections (i.e. Specify the Source Code by Generating a VIF). At this point the user is ready to begin the build process for his application. To accomplish this a VIF is generated at the request of the user. The user is free to view (or even edit) the VIF, but need not do so. This file is the primary input to the Custom Build Tool. In our case, the VIF will contain the file identifiers for Message_Spec, Message_Body_1, Buffer_Mgr_Spec, Buffer_Mgr_Body_1, Fast_Bus_Mgr_Spec, Fast_Bus_Mgr_Body_1, Router_Spec and Router_Body_1 in compilation order. The fact that the user is not necessarily aware of the contents of this file is important and cannot be emphasized enough. The user need only be aware of the class Message and the abstract notion of the instance Fast_Buffered_Messages which is fast and does not utilize a great deal of memory. When the VIF is generated from information contained in the

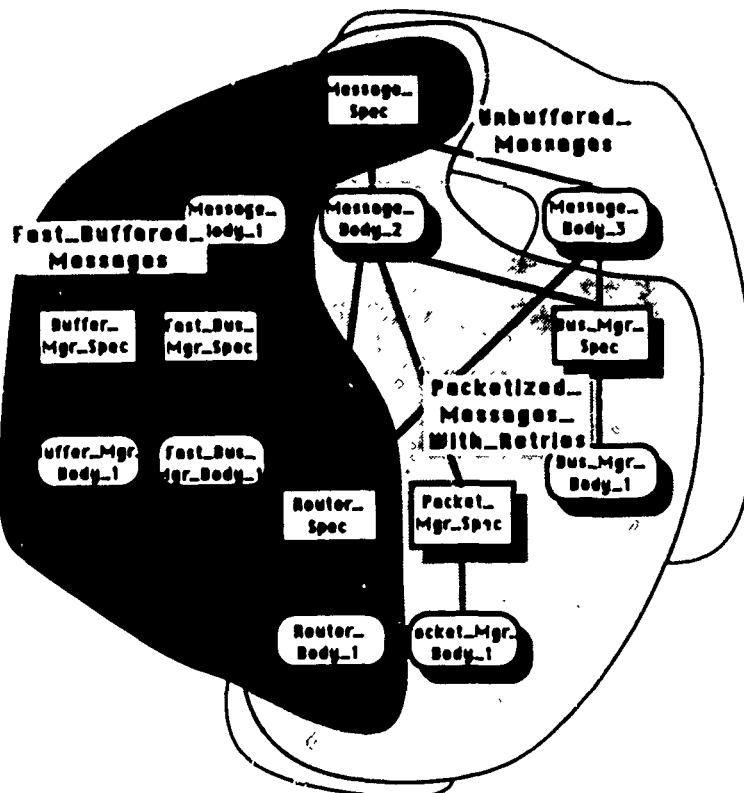


FIGURE 6. THE MESSAGE CLASS.

library, the details on how to construct the instance are automatically filled in for the user.

Step 6: Place the Selected Components Into the User's Library (i.e. Invoke the CBT). The CBT can either provide the ability to rebuild an ARTSE and compile user classes in the case of a cross compilation system target or just compile user classes for a host-host compilation system. The CBT parses the VIF and generates the ACS specific invocation necessary to insure the compilation into the appropriate Ada library of all code specified in the VIF. The user need only inform the CBT of the target Ada library. Once the CBT has been executed, the user can compile his application code against the new installed classes or link his application in order to create a program containing his tailored ARTSE. In our example, running the CBT will simply compile all of the specifications and bodies needed to implement the *Fast_Buffered_Messages* instance into his working library. Our sample user is then free to write the application code that will use the class *Message* specification and compile against it. When the application code is done, the user can link to obtain a load module containing the *Fast_Buffered_Messages* instance.

GENESYS AND THE SOFTWARE DEVELOPMENT PROCESS

As was seen in the prior sections, GENESYS is, in general, a reuse support tool and, specifically, a tool supporting the tailoring, as well as, the reusing of ARTSE and application components. To properly utilize GENESYS in a software application, the development team should supplement the normal software life-cycle process as early as the software requirements phase in of recognition the potential impact that GENESYS may have on analysis, design, implementation, testing and integration. Figure 7 presents a traditional "Waterfall" diagram

of the software development process annotated to show the effects GENESYS can have on the development process. The following paragraphs discuss the changes to the software development process required to take full advantage of GENESYS.

Figure 7 shows that during the software requirements definition phase, GENESYS should be considered within two contexts. First, when the requirements for the software system are being defined, the analysts must also consider the requirements of the Ada Run-Time Support Environment (ARTSE). Issues of potential importance include Ada Language Reference Manual (LRM) chapter 13 support and algorithmic performance characteristics of the ARTSE. The ARTSE requirement: play a key role in the selection of the production Ada compiler for the system.

Second, while defining the requirements for the system, the analysts should be able to identify the availability of GENESYS classes that already exist. With the knowledge of these existing classes, the analysts can specify the requirements in a manner that will not later preclude their reuse. To prevent restrictive requirements definition, the analysts should follow an iterative process of identifying the requirements, reviewing available classes and specifying the requirements. If the analyst finds one or more classes that may satisfy the requirements of the system, the requirements should be defined such that use of any of the classes is not precluded. Depending on the perceived closeness of fit, the requirements can be defined in such a manner as to, in fact, encourage the use of available classes.

During top level design -- definition of software interfaces -- the designers will identify the new GENESYS classes required for the application. As figure 7 depicts, there are two basic types of

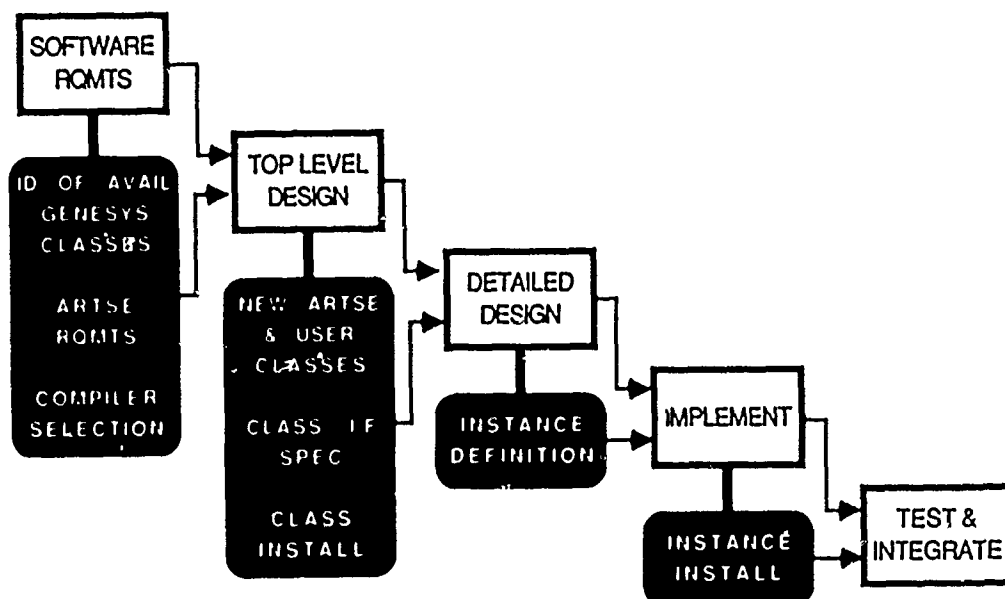


FIGURE 7. GENESYS IMPACTS ON THE DEVELOPMENT PROCESS

classes: ARTSE and USER. The new ARTSE classes to be defined are for those parts of the selected Ada compiler's ARTSE that require tailoring and for which no existing ARTSE classes currently exist. The same applies for USER classes -- classes that implement system level or application specific but not ARTSE functionality. Once these USER classes have been defined, their interfaces can be specified. Interfaces for ARTSE classes are already defined by the Ada compiler. However, the classes of run-time support which require tailoring must be added to (installed in) GENESYS. (A common interface standard for Ada run-time systems would eliminate the need to install new ARTSE classes each time a different compiler is used.) ARTSE classes can initially be installed with only their default (vendor supplied) instances. Tailored instances can be added later. On the other hand, USER classes can be new and therefore require specification of the interface as well as installation of the class. (Class installation is discussed above.)

Once development advances to the design phase, the designers are required to define the instances of the new ARTSE and USER classes. This process is generally no different than the detailed design of other software components. However, the development team should keep in mind that since the GENESYS USER classes tend to be system level (executive) or utility routines used by many other software components, the interfaces for these classes should be implemented early to provide the basis for designing other components. A final point in the detailed design of class instances: multiple instances for a class may be initially defined. The reason for the development of multiple instances is to provide alternatives with different characteristics that, depending on the actual final system and environment, may be optimum. This approach reduces the project and component development risk since it is likely that one of the developed instances will meet both component and overall system requirements. An example would be two instances, one that optimizes for speed and one that optimizes for memory utilization. If memory is constrained in the system and preliminary analysis shows that some components may need optimizing for memory, it may be desirable to design both instances. The goal would be to use the speed optimized instance unless memory dictates otherwise.

Implementation proceeds normally with one small addition. As class instances become available, they need to be added to GENESYS in the manner defined above. Once installed in GENESYS, they become available to all programmers for unit testing.

Since GENESYS actually performs the build process -- compiling the source code for the session instances into the user's library -- GENESYS directly supports the integration process. GENESYS also facilitates the testing of alternative implementations to assess their impacts on various system performance characteristics. By using different class instances the development team can

determine what affect substituting one instance for another has on total system performance. GENESYS manages the software configurations for the different class implementations (instances).

If your typical software development projects better fit the Spiral Model [BOHM86], where the phases depicted in the Waterfall Model are iterated N times, then the GENESYS tool should prove even more valuable. An iteration through the spiral can involve GENESYS at many different levels. At the least, GENESYS will be used to re-integrate the various software components. In a more comprehensive iteration, the selected Ada compiler may have been replaced with another -- requiring redefinition of ARTSE classes and instances to match the new compiler's run-time system. It is anticipated that the most frequent utilization of GENESYS in iterations through the spiral development process would be the implementation or modification of alternative instances and, less frequently, the definition of additional classes.

In summary, GENESYS supplements the traditional software development life-cycle in much the same manner as other reuse support tools with one very important addition. Although, some additional effort is required in the early stages, GENESYS should payoff over the entire life-cycle through:

1. Productivity increases through reuse, and
2. Reduced development risk through multiple instances.

The potential for risk reduction by supporting multiple development paths for the same general class of functionality, makes GENESYS a unique software reuse and tailoring tool for Ada development.

LESSONS LEARNED AND OBSERVATIONS

Our work on GENESYS has resulted in lessons and observations that can easily be categorized as development support system issues and GENESYS technology issues. First a discussion of the latter.

STANDARD ARTSE INTERFACES. Because run-time interfaces are not uniform across different compilation systems, a Custom Build Tool (the part of GENESYS that performs the compiler specific application build) must be developed for each unique set of ARTSE interfaces. Acceptance of an ARTSE interfacing standard such as the proposed ARTFWG framework would be a big step in, not only increasing the utility of GENESYS, but of opening embedded systems to greater tailorability, portability and reusability.

INSTANCE SELECTION COLLISIONS. Because the GENESYS class-instance structure is very flexible, it is possible for more than one instance to utilize the same Ada package specifications and body alternatives. It is possible that two different body implementations from separate selected class instances may be included in a user's session.

This would result in a system that does not function as expected since the last body compiled would be the one used by both classes. Currently, this issue is not addressed by GENESYS. Potential solutions include: prompting the user to select only one of the bodies or modification of the source code to allow use of both instances. The former solution is obviously the easier of the two.

CLASS-INSTANCE MODEL SUPPORTS TAILORABILITY AND REUSE. We quickly discovered that the class-instance model we structured to support ARTISE tailorability was equally applicable to support the tailoring of any domain of software. In retrospect, this should have been expected since one of the desirable attributes of object-oriented programming languages, such as Smalltalk, is the inheritance mechanism which supports tailoring while encouraging reuse.

CLASS-INSTANCE MODEL ENCOURAGES INTERFACE STANDARDIZATION. By using the class concept to encapsulate abstractions, GENESYS helps to enforce the standardization of software interfaces while permitting flexibility in implementation. Using abstract data types as an example, a standard interface to a stack class could be defined. Several instances for the stack would then be implemented following Booch's taxonomy [BOOC87]. The different instances, implementing bounded, unbounded, concurrent, nonconcurrent, etc. variations on the stack data type, would all have the same interface for users of the class.

CLASS-INSTANCE MODEL CAN HELP REDUCE RISK. By concurrently developing multiple solutions to the same class of functionality, projects can reduce the risk of any one solution will fail. The class-instance model employed by GENESYS, facilitates this risk reduction by supporting multiple instances for the same class. If at any time during development, any one instance is shown to fail, one of the remaining instances can be used. During final testing, the instance that best provides the required functionality can be selected. This paradigm encourages the development of innovative solutions in conjunction with reusing tried solutions or developing customized versions of tried solutions without placing the project at unacceptable risk.

The development support systems issues include three specific areas: Ada compiler and support development tools, X windows and X Ray toolkit and relational database management system (RDBMS).

ADA COMPILER. The Alsys Ada compiler for the Sun 3 family of workstations was generally adequate. An interesting observation was that large source files that were compilable with the Alsys compiler on an IBM PC-AT were too large for the Sun version. Occasional problems with the program libraries occurred. The problems resulted in an unusable library which required the recompilation of all code. It was noted that if a compilation was aborted by the user (control-C), the user took the chance of corrupting the program library.

X WINDOWS AND THE X RAY TOOLKIT. The X windowing environment provides a portable window and graphic environment. However, the X 11.2 distribution from MIT suffered from performance problems. The X environment is made easier to program through a toolkit. We employed the X Ray toolkit since we had available an Ada binding for it from SAIC (also a STARS Foundation product). The X Ray toolkit had poor documentation and the Ada binding had several bugs. (It should be noted that SAIC did an excellent job in making the binding available quickly for the rest of the STARS community and in supporting other STARS contractors.) Finally, the X Ray toolkit did not provide widgets for the types of window/graphical objects that would have been better suited for the GENESYS user interface. If time permitted, we could have written our own widgets.

RELATIONAL DATABASE MANAGEMENT SYSTEM — ORACLE. Use of the Pro*C programmer's interface to the Oracle RDBMS was successful. The only problems encountered were the usual nuances associated with a misunderstanding between the documentation and the programmer's interpretation. Surprisingly, no serious problems were encountered in using the Oracle RDBMS, C subroutines and the Ada run-time environment.

REFERENCES

- [BOCH86] Boehm, B.W., "A Spiral Model of Software Development and Enhancement," Proceedings IEEE Second Software Process Workshop, ACM Software Engineering Notes, Aug 1986.
- [BOOC87] Booch, G., "Software Components with Ada," Grady Booch, Benjamin/Cummings Publishing Co. Inc., Menlo Park, CA, 1987.
- [BUR87] Burton, B.A., Aragon, R.W., Bailey, S.A., et al, "The Reusable Software Library," IEEE Software 4(7), pp. 25-33. July 1987.
- [FALA88] Falacara, Gary, Angevine, Mark, Bailey, Stephen, and Laird, Jim, "A Tool for Ada Run-Time Tailoring," Proceedings AdaExpo '88, Oct 88.
- [STRO88] Stroustrup, B., "What is Object-Oriented Programming," IEEE Software, May 1988, pp. 10-20.

BIOGRAPHY



STEPHEN A. BAILEY

Intermetrics Inc.
Development Systems Group
4733 Bethesda Avenue Suite 415
Bethesda, MD 20014

Stephen Bailey is Manager of Environment Development and Integration. His responsibilities include research, development, integration and production of software development tools and environments. Specific areas of interest include software reusability, productivity, graphical/windowing user interfaces and distributed environments. Mr. Bailey received his B.S. and M.S. in Computer Science from Chapman College.



GARY N. FALACARA

Sparta Inc.
P.O. Box 909
Burbank, CA 91503

Gary Falacara is a senior software specialist engaged in the development of methodologies and tools for software engineering and in Ada software development for real-time systems. His interests include operating systems, distributed systems, computer architecture and software engineering methodologies. Mr. Falacara received a B.S. in Applied Mathematics from N.C. State University and a M.S. in Computer Science from California State University at Northridge.



JAMES L. LAIRD

Intermetrics Inc.
Aerospace Systems Group
5312 Bolsa Avenue
Huntington Beach, CA 92649

Jim Laird is the business development manager for military space systems. He is responsible for applying state of the art Ada technologies to embedded, spaceborne, real-time systems. His interests include operating systems, distributed systems and real-time software engineering methodologies. Mr. Laird received his Bachelor of Science from the University of California at Irvine.

MARK ANGEVINE

Aerojet Electro-Systems Company
1100 West Hollyvale Street
Azusa, CA 91702

Mark Angevine is a programming staff specialist engaged in the research, development and evaluation of software development methodologies and tools. His interests include software engineering, Ada, parallel programming methodologies and compiler technology. Mr. Angevine received a B.S. in Biology from Yale and a Ph.D. in Population Biology from Cornell.

An Ada Implementation of the Data Encryption Standard in a Real Time Environment

Mr. Larry Grosberg & Mr. David Coe

Advanced Software Technology Area
of the U.S. Army, GECOM, Ft. Monmouth, N.J.

ABSTRACT

Many studies involving the Ada programming language rely on simplistic examples to incorporate a solution space. The given study is based on data security, considered to be a high priority with many software engineering researchers. The DES (Data Encryption Standard) was the major focus of the analysis.

1. Introduction

The necessity to use cryptography in order to protect stored and transmitted data from intruders and eavesdroppers has been recognized in many applications such as electronic funds transfer, automated clearinghouses, and securing non-official computerized military data.

In 1973, the U.S. National Bureau of Standards responding to public concern about the confidentiality of computerized data outside military and diplomatic channels, invited the submission of data encryption techniques as the first step toward an encryption scheme intended for public use.

The method selected by the bureau was developed by IBM researchers. Known today as the Data Encryption Standard, it was issued in January 1977 as the Federal Information Processing Standard Publication (FIPS PUB) 46. It is also known as ANSI standard DEA since 1981 (ANSI 1980) and it is already in use in many industrial applications. In recent years, the DES has been proposed as an ISO (International Standard Organization) standard under the name of DEA1 (ISO, 1983).

Federal agencies and departments needing such protection can purchase commercial DES implementations that have been validated by the National Bureau of Standards as conforming to the standard. Software implementations do not comply with the standard and are generally inefficient as compared to the hardware versions. The software implementations can still provide adequate support to many hardware systems in the cases where problems with the cryptographic unit results in a loss of integrity of the encrypted data. It is clear that although the step-by-step DES algorithm is available in the public domain, the mathematical reasoning behind the DES algorithm is considered confidential by the NBS.

2. Project Scenario

In the context of this report, the DES system will provide cryptographic support to a number of "in-flight" rockets (via a DES integrated chip). The direction and position data of each rocket is required to be secure from eavesdropping by the potential enemy. In addition, the DES system will incorporate a software support mechanism, whereby the integrity of the DES system can be verified. This requires that the DES software support mechanism provide encryption/decryption operations in a "stand alone" fashion when severe hardware failures occur during the communication linkage of the "in-flight" rockets.

3. Scope

The DES system was implemented in the Ada programming language. The hardware and software implementations focussed upon an "Ada only" philosophy throughout the development life cycle of the project. The goals of the project include:

- * Verify the design impact that Ada has on program development using the Ada implementation of the DES algorithm as a complex model.
- * Develop an appreciation for the performance issues involved in the development of Ada real-time projects.
- * To incorporate an Ada in-line code emulator to understand how it can be used in the development of Ada distributed systems.
- * To gain insight into the hardware implications that pertain throughout Ada project developments.

4. Analysis of The DES Algorithm

The DES is a single key system in which data is both encrypted and decrypted with the same key, a sequence of eight numbers, each between zero and one-hundred-and-seventeen. The algorithm divides a bit message into blocks of eight characters, then enciphers them one after another under the control of a sixty-four bit key. The letters and numbers of each block are scrambled no fewer than sixteen times resulting in eight characters of cipher text.

The DES is immune to brute force attacks since it would take a machine computing one million trials per second over a millennium to cover all of the 2^{56} possible keys. IBM and the National Bureau of Standards warn against employing around 200 of the DES's keys since those keys are considered semi-weak keys. A semi-weak key is any key that might create clues in an encrypted message that could lead to its decipherment in less time than a brute force attack would consume.

5. Development Background

The development method utilized in this project is that of the "Bottom-Up" method of software design. In this methodology, the lowest level modules are the ones to be designed and coded first in the development. Succeeding modules are then designed in a hierarchical fashion until the progression towards the main module is complete. This methodology was chosen because of the abundance of independent low-level modules (without much up front design overhead) that were required to be implemented in the early phases of the DES system project.

5.1 Software Development Background

The software DES subsystem is based upon the standard DES algorithm for motivation in its implementation. The most important cryptographic function employed by the DES algorithm is the product transformation. It consists of successive applications of substitution and transposition ciphers. Transposition ciphers involve an encryption procedure that changes the normal pattern of the characters in the original plain text message. Substitution ciphers on the other hand, replace blocks of characters with substitutes.

5.2 Hardware Development Background

The target system is an Intel Integrated Circuit Emulator unit (ICE), where the Ada code is generated on a MicroVax II using the DDC-I 8086 Ada cross-compiler.

The ICE unit was selected as the target to facilitate the optimizing and debugging of real time Ada code. The Digital Encoding chip (WD20C03A) is connected to a serial port on the ICE unit.

The Western Digital device will be programmed to use the Cipher Block Chaining mode to provide security to the system's transmissions. Once the Digital Encoding chip has been initialized, the system is then ready to encrypt or decrypt messages. The Western Digital Encoding/Decoding chip necessitates that its data register (including the application) receive one byte of data at a time in groups of eight. This requirement is achieved by the use of the function `Unchecked_Conversion` which converts the bit pattern of the source to that of the target (utilizing the same amount of memory). These messages may originate from the rocket subsystem or from the rocket command subsystem.

6. System Development

The development of the DES system incorporates the hardware and software implementations of the DES standard. The duality between the hardware and the software DES subsystems allows the overall system to anticipate a high degree of integrity. Thus a task BIT (Built in Test) will check the operational integrity of the primary mode of the DES system (i.e. hardware) to that of the secondary mode (i.e. software). Obviously, the software mode will offer the customer (i.e. "in-flight" rocket) a decreased throughput that is directly proportional to the number of customers that are requesting service in the given time interval.

6.1 Software System Development

The software subsystem consists of four Ada packages (about 900 lines) providing direct and indirect support to the application DES module. The software subsystem is designed to handle words (i.e. 16 bit values) in the range of -32767 to +32767 of `bam_type`. (Arbitrarily classified as `bam_type` but identical to the integer type found on many P.C. implementations) All data must be of the said type (or converted), before the given data can be processed. The encryption key, External or Internal, must be incorporated into the system before the key can be processed, and thus before any encryption can occur. Note that the data to be encrypted is considered to be processed in a sixty-four bit envelope (i.e. four words) with zeroes being employed as padding if the submitted data envelope falls short.

6.1.1 Development Structure

Package Rocket_Types => (50 Lines) defines the major data types that are to be employed by the rocket scenario model.

Package Utility_Package => (83 Lines, 2 procedures, 3 functions) consists of specified DES types and atomic modules that are to be employed by the DES architecture.

Package SW_DES_Package => (201 lines, 4 procedures) the upper level package that provides encrypt/decrypt DES functions in a variety of envelopes.

6.2 Hardware System Development

The first module to be designed was the procedure responsible for the initialization of the WD20C03A. The receiving and sending of information to port addresses was accomplished through procedures in the DDC-I's Low_Level_IO package, Receive_Control and Send_Control. (Since there are no built-in features in Ada to handle bit manipulation, a procedure was designed to handle this requirement using the Ada Machine_Code package)

In the next level of design, there is a task which allocates the DES resources for either guidance or position data. After insuring the first call to the task is for an encryption, the task loops indefinitely allocating resources for either encrypting or decrypting.

The next level is the Built-In-Test, BIT. This level supplies the integrity testing of the hardware Digital Encoding chip by verifying the hardware's results against those derived from a software emulation of the hardware's algorithm. At a pre-defined interval, BIT will test hardware integrity. If an error is detected, BIT will switch requests for encryption / decryption to the software emulation package.

7. Status of Development

7.1 Status of Software Development

The Software DES Subsystem has been tested against a detailed step by step example (see Katzan, 86). The time and effort required to develop "In-House" multiple test examples was found to be too costly in both respects, instead a simplification in the testing process was implemented. Many variations of the sixty-four bit text and/or key were tested. The assumption being that if the software subsystem can decipher the previous cipher message, then the system can be deduced to have some degree of integrity (only if the system was proven to be working against a known example as is the case).

The software DES subsystem provides an encryption time of 0.679 seconds for sixty-four bits (four words) of data. Encrypting the position data for five rockets took 3.46 seconds, and encrypting the guidance data for the five rockets took 3.67 seconds. (The MicroVax II and the DDC-I compilers were utilized). P.C. versions were also implemented which resulted in a decrease in performance levels as expected. (22% slower on the PC-XT)

7.2 Status of Hardware Development

A few inconsistencies with the DDC-I Ada compiler have resulted in the hardware being untested.

These inconsistencies are:

I. When the code for a certain package was enclosed in a single file, the compiler stated that the library was too small and that a new sub-library needed to be created. This error message occurred when writing the object code into the library. The same code with the modules in separate files compiled successfully into the library.

II. The use of Ada generics resulted in the inability to compile certain procedures needed for the terminal driver (inability to distinguish generic packages of overloaded procedures). Compiler stated the second packages' procedures already existed in the library.

III. The DDC-I compiler defines "Byte" to be a "new integer", which states that "byte" will take up two bytes of memory instead of one byte as expected. This caused problems with some of the procedures in the Low_Level_IO package.

As a side note, since the target system did not come with a monitor/keyboard for input/output during program execution, a terminal driver has been written using the DDC-I's Terminal_Driver package.

8. Conclusion

Once the system is executing as designed, the current target system can become a test bed for more experiments in the real time arena of Ada applications. The attitude of this group has been to unravel these unforeseen problems rather than by-pass them. This attitude allows programmers to observe the limitations of Ada in the real time environment. Obviously this is the inception of this project, the real significance will come when the system is ported from a single processor system to a multiple processor system with minimal restructuring effort.

9. Further research

The area of porting a program structured for a single processor system to one of multiple processors has numerous possible utilizations in the real time field. Also, the inconsistencies in the DDC-I Ada cross-compiler can be analyzed for definitive causes, at present there can only be suppositions as to the actual causes. In addition, the software encryption modules can be utilized in the field of Ada benchmarking. The complexity of the DES algorithm does incorporate many basic and complex Ada features. The vision of standard benchmarks that rates certain Ada environments by their respective DES throughputs in cps (encryptions/sec) can be of a valuable asset toward the evaluation of such environments.

Lawrence Grosberg is a member of the software engineering staff for the Advanced Software Technology area of the U.S. Army Communications Electronics Command Fort Monmouth, New Jersey. He holds a B.S. in Electrical Engineering and a M.S. in Software Engineering from Florida International University, Miami Florida and Monmouth College, West Long Branch New Jersey.

References

United States Department of Defense, Reference Manual for the Ada Programming Language MIL STD 1815A, Ada Joint Program Office, March, 1983.

Katzan, Harry. The Standard Data Encryption Algorithm. Prentice-Hall, 1985.

Bosworth, Bruce. Codes Ciphers and Computers. Hayden Book Company, Inc. Rochelle Park New Jersey 1982.

DDC-I Ada Cross-Compiler Reference Manual, Appendix F.

Acknowledgments

Dr. Thomas Wheeler, AMSEL-RD-SE-AST, Ft. Monmouth, N.J.

Mr. Tom Griest, LabTech Corporation,
Ms. Judy Richardson, AMSEL-RD-SE-AST,
Ft. Monmouth, N.J.

Mrs. Mary Bender, AMSEL-RD-SE-AST,
Ft. Monmouth, N.J.

Biographies

David Coe is a member of the software engineering staff for the Advanced Software Technology area of the U.S. Army Communications Electronics Command Fort Monmouth, New Jersey. He holds a B.S. in Electrical Engineering and a M.S. in Software Engineering from Widener University, Chester Pennsylvania and Monmouth College, West Long Branch New Jersey.

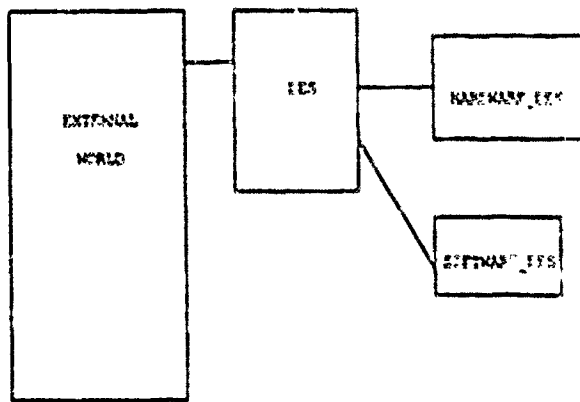


FIG. 1 DES CONTROL SYSTEM

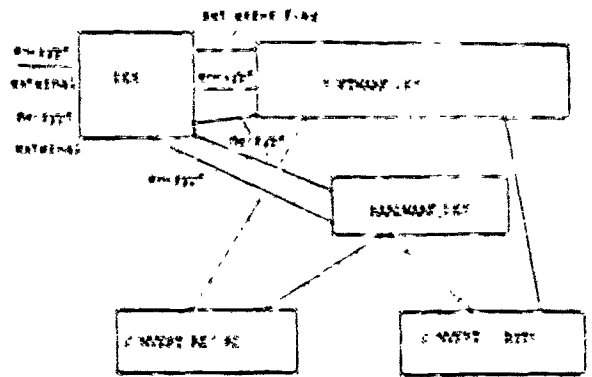


FIG. 2 DES-SYSTEM MODEL

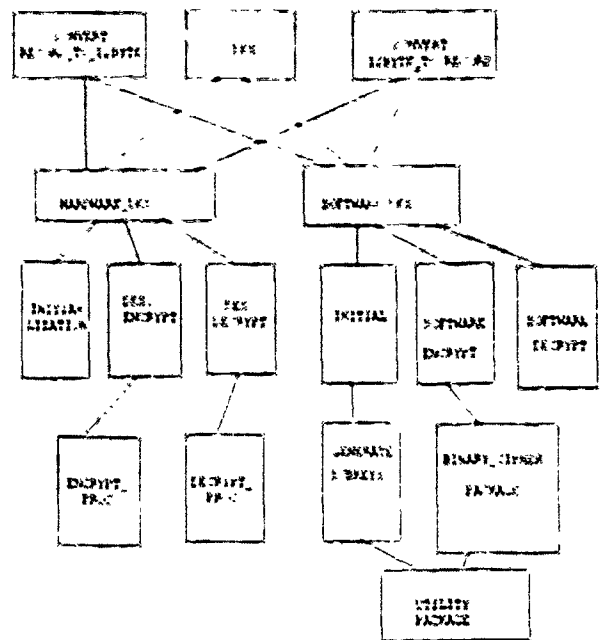


FIG. 3 DES-SYSTEM STRUCTURE DIAGRAM

A Hardware Independent System Development Approach Involving Ada

Tom Dale

Unisys Defense Systems
McLean, VA

Abstract

This paper presents lessons learned from a C3I project employing rapid prototyping and object-oriented programming. The project goal, to develop a hardware-independent prototype implementation of an advanced text (aka message) handling system, necessitated a software development strategy consisting of X Window, POSIX, NFS/TCP/IP/Ethernet, SQL, Ada, and C. This paper relates lessons learned about rapid prototyping, reusability, attaining hardware independence, designing a distributed architecture, using Ada in a multilingual environment, tuning the performance of Ada application code, and integrating COTS/NDI within an Ada environment.

1. Introduction

Ada has gained increasing acceptance for usage in non-embedded mission-critical applications. However, Ada is not the only "standard" currently receiving Government acceptance. A plethora of other standards have also received certain notoriety, such as Network File System (NFS), Structured Query Language (SQL), Portable Operating System Interface (x) (POSIX), Ethernet (IEEE 802.3), along with others.

Non-embedded mission-critical systems, in attempts to reduce system costs during these times of lean budgets, will employ commercially-available hardware and software where feasible. Attainment of hardware-independence further requires adherence to a strict suite of interface standards. This paper describes the engineering approach of an application in which Ada functions as one element of a hardware-independent approach.

This paper addresses concerns arising from developing a working prototype text handling environment consisting of Commercial-Off-The-Shelf (COTS) software and newly developed code written in both Ada and C. This limited domain description parallels work underway at the Software Engineering Institute (SEI) regarding domain-specific architectures but differs in origination, intent, emphasis, and scope.

SEI currently investigates message handling environments to identify "recurring problems." These recurring problems are evidenced by assigning functions to threads of control and noting those functions represented on more than one thread.

This paper describes an independent industry attempt to develop a methodology for quickly delivering working prototypes into the marketplace. In 1987, Unisys Defense Systems originated an IRAD (Independent Research and Development) project to develop technology for quickly assembling text handling environments to suit a wide variety of users. Message handling domain experts felt there existed a better way to develop message handling environments. The project's intent was to quickly develop a working prototype satisfying military requirements, not to produce a strictly Ada solution. This work may be characterized by its implementation emphasis on 1) ergonomics with system requirements developed "from the user interface on back" in an integrated application environment and 2) development of reusable components designed to address specific implementation concerns regarding performance, scalability, and configurability. The scope centered on providing a modular design that could incorporate emerging technologies (i.e. permit easy technology insertion).

2. Business Rationale

This IRAD project is aimed at effecting technology insertion by developing text handling components capable of incrementally replacing existing components. Unisys Defense Systems based its rationale on the fact that the prototype must be integrable into an existing system, and to provide better service for less cost. A superior product (working prototype or not) satisfying mission needs not only enhances business perceptions, but also provides users a greater capability and user project management a success story. Tighter budgets pressure the military to replace obsolete, expensive to operate systems with modern hardware and software which will increase performance and cut costs.

Unisys Defense Systems realized the importance of a comprehensive strategy for integrating different components. Open Systems are software environments comprising products and technologies that are designed and implemented in accordance with vendor-independent, commonly available standards.

Ada will change the way companies do business with DOD because the language is the same from one system to another. Across-the-board use of Ada provides the competitive edge to the company with the best overall engineering solution. Separate initiatives within DOD to use COTS products and industrial standards wherever possible also reduces the potential for unnecessarily restrictive competition. It seems reasonable to incorporate strengths from both initiatives.

3. Approach

Unisys Defense Systems utilized an approach predicated upon the development of a hardware independent toolkit of text handling components.

Text handling conforms, to a large extent, to a dataflow paradigm. Problem decomposition, a central design feature of any large system, whether prototype or production, yielded a top level functional decomposition (ala the structured analysis school of thought) which provided a system view of the interconnecting building blocks. The system's working scenarios then provided system requirements, operations concept, and an initial thread design. Those threads capable of utilizing reusable components were selected first for prototyping. This strategy focused the prototype effort on successively demonstrating implemented threads.

3.1 Rapid Prototyping

Dynamic and transaction processing oriented systems involving extensive user dialogues tend to be the best applications for rapid prototyping. A rapid prototyping methodology allows iterative refinement of system requirements and embedding as many of these requirements as possible into the user interface to build the system from the user interface backwards. The system's objects resulted from this iterative prototyping of the user interface.

Two major types of life cycle models for prototypes exist. In one, the prototype is regarded as a throwaway to be discarded when the real production system is implemented. In the other, portions or all of the prototypes wind up as the end product, actually becoming the final production system. The goal of a prototype typically differs from that of a production software system in that effective use of designer time and rapid user feedback have greater importance than efficient use of machine resources, completeness, and robust operation. However, given a competitive business environment without the luxury of time, business strategy dictated development of a prototype system capable of replacing existing production systems.

No sensible manager would commit to rapid prototyping armed only with third generation compiled languages and some batch data management utilities. To put the "rapid" in the prototype, be prepared to evaluate, select, and purchase some software development tools that could be quite expensive.

Innovations promising big payoffs are also accompanied by a certain degree of risk. To sell management on the benefits, you must also be in a position to spell out precisely how you intend to control and minimize those risks. A distinction should be made between change management and configuration management. Change management tracks the changes to each individual component of a system. Configuration management adds the capability to organize, manage, and track all pieces of an application as a unit.

Only one prototyped function could not become part of a final system. One function was prototyped using SUNVIEW since at that time a requirement existed for a window-based application and X11 was not stable enough to use.

Prototype elements have been evaluated to identify potential difficulties in deliverable versions. Statistics concerning the firing frequencies of the prototype's components identified potential performance bottlenecks.

User feedback helped to evaluate the appropriateness of the prototype design concepts.

3.2 Reusable Components

A different perspective on reusability permits design of parameter-driven functions as reusable software components which need not be strictly Ada generics. Generics are used where applicable to effect 'Ada' reuse (e.g. doubly-linked list generic); however reusable software components reduce both development and maintenance risks. These reusable software components correspond to operations performed in the system and reduce both development and maintenance risks.

Dataflow and control flow are two popular decomposition criteria. Components of a dataflow decomposition are independent sequential processes that communicate through buffered data streams (essentially FIFO queues), while the components of a control-flow decomposition are procedures that are called by and return to a main procedure with a single control thread.

High-quality reusable text handling components are attainable. It is important to have a relatively complete set of general-purpose components to perform the functions common to many systems, such as managing displays, sorting and searching, parsing input strings, and managing look-up tables. Many of these functions can be effectively encapsulated in a small set of abstract data types. It is very important to provide generic versions of the reusable components because it would otherwise be impossible to design with abstract data types while relying on standard reusable components for performing common utility functions.

A strategy based on reusable software components is a promising, practical approach to rapid prototyping. Modularity is especially important in prototyping because of the need to make many changes in a short time. A systematic method for prototyping is necessary but not sufficient for the rapid construction of prototypes for large real-time systems.

The engine concept (atomicity) evolved. An engine is not machine-specific. A specialist works on an engine which allows creation of new features without having to split work among multiple teams. This method avoided the tracking of changes required should a layering approach based on different operating environments have been used.

3.3. Distributed Architecture

When designing a distributed architecture, top level functional decomposition allows a designer to obtain a system view of the interconnecting building blocks describing functional (service) requirements. Resulting modularity increases productivity by reducing debugging efforts and improves understandability, reliability, and maintainability of developed system software, features especially important in rapid prototyping.

The number of modules affected by a change is limited and can be determined by a straightforward mechanical analysis of the prototype's dataflow structure and thread delineation. Distribution of computational parts among several processors becomes easier, since implicit interactions, difficult to implement in a loosely coupled architecture, have been eliminated.

The trend exists toward distributed applications based on the client-server model. When multiple users access a common resource, performance becomes a critical issue. Distributed processing is key to maintaining top server performance. The processing potential of each network node should be exploited instead of letting the server do all the work. One solution to this problem is to use a distributed environment based on low-cost workstations. Rapid advances in computer hardware have allowed networked PCs to become an alternative for many military applications once reserved for mainframes or minicomputers. High-speed 80286- and 80386-based PCs have the raw computing power of minicomputers.

Interprocess communications also allow applications to send messages to each other based on the results of specific actions. Servers were connected (whatever their source language) by communicating through files. Two processes must agree on the interchange protocol which adds complexity but allows modularity. Eventually, such integration code will be performed by knowledge-based engineering tools.

3.4 Standards

Standardization and openness are taken for granted in telephones, telecommunications switches, televisions, radios and compact disk players. Whoever the manufacturer, all these devices can "talk to each other" and freely interconnect.

Standards allow users to move forward with technology innovations while protecting software investments and minimizing training time. Standards allow software developers to concentrate on writing applications and solving problems at hand. Standards provide transportability across various platforms, thereby protecting a software developer's investment.

Standards reduce the risk of obsolescence and reduce costs because they're supported by multiple vendors. They also reduce training, service and support costs. Costs associated with software re-writing, staff re-training, and loss of access to information are monumental when users are required to move from one proprietary system to another. In fact, the likelihood is slim that a user can even maintain and reuse all his or her information -- years of collected data -- after a major change in hardware.

Users have demanded standardization to hook together heterogeneous computing environments so that application programs can move between different operating systems within a single network.

Once standards are accepted, hardware comparisons become easier. Standardization reduces product uniqueness; it can also increase the return on research and development investment. As the software-hardware interface becomes more standardized, software developers do less work to get more money. Fewer channels will be needed for marketing, and the marketability of many software products will be enhanced as software applications begin to run on more types of hardware and become marketed through wider distribution channels.

Software should be purchased from vendors who are viable for the long term and have solid migration strategies. Since these vendors provide a migration path as standards evolve through multiple permutations, they can spread the burden of migration rewrites over the entire customer base.

Both purchased and internally-developed software should be written in languages that are likely to continue as standards. In addition, code generators should be selected which generate standard languages.

This rush to provide users with the standards they demand such as POSIX, SQL, and Ada does not in itself provide the foundation for viable open systems. Vendors must provide these standards in an integrated and common fashion that assures consistency of implementation. Standards are defined with allowances for implementor-defined options, portending different "standard" variations.

3.4.1 X11

In distributed environments, the X Window System, or X11, developed at the Massachusetts Institute of Technology (MIT) and supported by the X Consortium, enabling most workstation manufacturers, offers network transparency and unprecedented portability of applications programs. Applications running on central mainframes, minicomputers or other workstations can display results on any vendors' local workstations running the X11 server.

In X11, the connection is made over the network. As long as an application is capable of issuing X11 protocol requests over the network, it can display output on, or obtain input from, any device on the network that is running an X11 server.

Since the major workstation vendors have committed to supporting X11, system integrators can avoid being locked into writing their applications for a proprietary windowing system environment and can choose workstation hardware from any vendor to support their applications requirements.

The X11 client application is made of several layers: the Xlib programming interface, which interfaces with MIT toolkits, which can interface with high-level graphics (e.g. PHIGS, GKS), from which a look and feel can be developed. X11 doesn't specify the look and feel of the user interface; it simply provides a set of tools with which to build one. The X Toolkit, which includes X Toolkit Intrinsics and X Widgets, contains software tools for X-based applications development. The look and feel consists of how the graphics and icons align on the screen and how the toolkit gets called. Ultimately, a user interface needs to be intuitive to be valuable, with "intuitive" meaning menu-based or icon-based. The comfort factor of the user with an interface is what determines acceptance.

The client requests graphics, and the server provides them. The server can be local or on a different type of CPU on the other side of the building; the user doesn't have to know. This allows applications to run on platforms where they can run best, rather than being restricted to the workstation.

3.4.2 POSIX

The term POSIX is an IEEE trademark. POSIX aims at making the independent part as large a proportion of the whole as possible. McCarron [1] provides an excellent compendium of POSIX, parts of which have been paraphrased freely and included herein. Most vendors will meet POSIX standards, albeit some grudgingly.

The issue is not whether we will reach POSIX compliance but how quickly we will reach POSIX compliance. POSIX attempts to effect software, hardware, maintenance and training savings. The Government has assumed the lead in open systems which will return value as new applications and technology become available. The Government desires "all computers to work in multivendor environments. This goal consists of three parts: portability, interoperability, and scalability. Portability enables systems - even those from different vendors - that meet POSIX specifications to use the same application software. Interoperability allows the computers to work together, and scalability means that different sizes of computers - from personal computers to supercomputers - can exist in the same software environment.

The POSIX IEEE 1003.1 standard is a Unix-based operating system interface designed to provide portability of applications software at the source code level by producing a System Services Interface standard. P1003.1 defines a set of system calls and library routines, some of which are optional. It does not address operations such as user commands.

NIST created the initial POSIX Federal Information Processing Standard 151 (FIPS 151) to give government buyers a set of specifications to use before IEEE formally adopted 1003.1 from which POSIX FIPS 151 differs in small substantive ways. Many of these differences are generational since the FIPS is based on Draft 12.0 of P1003.1, while the final standard is based on Draft 13.0. NIST is now committed to revising the FIPS and bringing it into line with Draft 13.0. P1003.1 is expected to be adopted by ANSI and ISO making it an international standard in 1989.

The IEEE standard offers some options mandated in the FIPS. In some instances where the full-use standard offers a choice for implementations, NIST has selected one choice to ensure application portability. The resulting FIPS requires vendors to meet a more demanding specification. This unilateral move by NIST on POSIX is a departure from its tradition of waiting for vendors to adopt standards before drafting federal versions of those specifications.

NIST has another effort underway. With the help of X/Open, it has developed a proposed family of next-generation standards called the Applications Portability Profile (APP). The APP addresses POSIX bindings, or links between applications and POSIX specifications, for a host of functions, including: database management; data interchanges for document processing, graphics and product data; network services for data communications and file management; and hooks to support different computer languages (C, COBOL, Pascal, FORTRAN and Ada).

The IEEE POSIX committee has recently formed a subcommittee, called the P1003.0 group, chartered with defining a portable operating environment, similar to the NIST effort. P1003.0's objective is to integrate various application standards (windowing systems, communications, programming languages, database access, graphics and user interface) with POSIX and each other to create a public domain open systems environment as robust as a proprietary system environment.

P1003.2, Shell and Tools Interface, defines a programmatic interface for shells, tools, and some commonly found Unix utilities like awk, grep, lp, yacc, etc. The command set was frozen in March 1988 with the standard shell and tool interface expected late in 1989.

P1003.5, Ada Binding for P1003.1, will develop language-independent representations of each service described in P1003.1 so that Ada representations may be established. P1003.1 is based on C.

Other POSIX efforts include P1003.3, Testing and Verification, P1003.4, Real Time, and P1003.6, Security.

POSIX and the System V Interface Definition (SVID) test suite for Unix System V compatibility overlap. Roughly 30 percent of the SVID specifications are not included in POSIX, while about 50 percent of POSIX is not found in the SVID. AT&T plans to make release 4.0 of Unix System V --which is due late next year--POSIX-compliant.

3.4.3 NFS

NFS provides transparent, remote access to filesystems. NFS uses an External Data Representation (XDR) to describe its protocols in a machine and system independent way. NFS is implemented on top of a Remote Procedure Call (RPC) package to simplify protocol definition, implementation, and maintenance since RPCs are synchronous. NFS uses a stateless protocol to facilitate crash recovery. NFS does not support all of the Unix semantics.

It's not uncommon to find a network with 10 to 12 workstations on Ethernet being slowed down due to bandwidth limitations, making suitable NFS performance questionable. An Ethernet can be divided into smaller departmental systems with a faster, broadband or fiber cable connecting them these communities of interest. Such "internetworking" will be a key issue for network managers in the near future as well as long term. A Fiber Distributed Data Interface (FDDI) network has transmission speeds of 100 Mbps and, since the token is released immediately after transmission, supports circulation of multiple messages on the network simultaneously.

3.4.4 SQL

SQL, developed for relational databases, represents a method to manage and query relational databases. SQL is implemented on personal computers, minicomputers, and mainframes. Most vendors have developed SQL supersets. More recently, vendors have implemented distributed versions of their SQL systems. Essential integrated development tools have been implemented by virtually all vendors to support relational database management system (RDBMS) application development.

Dr. E.F. Cobb invented RDBMS over 20 years ago at IBM. In 1985, he published 12 rules defining RDBMS in an attempt to keep the term "relational" from being corrupted by database vendors. The ANSI SQL standard falls short of recommending a RDBMS as defined by Dr. Cobb. For example, ANSI SQL does not offer any recommendations for a catalog or indexes although most vendors do offer this capability.

Most vendors are working on extensions to their RDBMSs far beyond that envisioned by Dr. Cobb or those recommended in the ANSI standard. Vendors are now implementing commands for the manipulation of complex databases including digitized voice, bit-mapped graphics, memoranda, and others.

3.4.5 Ada

Ada has been mandated for use in developing new systems. The term "Ada" includes not only the Ada language but also the environment, software engineering, good software tools, configuration control, etc. However, anticipated cost savings from the use of Ada are not being realized.

Ada packages are a good means of implementing standard components. Ada packages allow a superior design approach since packages can encapsulate both data and procedure resources required to implement a client/server mechanism.

The Ada tasking model determines how and in what sequence program tasks are performed. This model was designed on the assumption that one executable Ada program would simultaneously execute many tasks. Weaknesses of the Ada tasking model are well-known [2]. With the Ada tasking model, users cannot clearly specify particular times or priorities in a program. Although this operation can be theoretically achieved, the implementation typically creates a large system overhead that slows down performance.

Ada pragmas allow data sharing among programs and provide interfaces to procedures written in other languages, such as Fortran, C, or assembly. This capability gives users access to operating system facilities and supplies a convenient method for incorporating real-time features in an Ada system. At the same time, this methodology fulfills the letter of the Ada language requirements but maybe not the spirit.

Ada suppliers now provide other methods to deal with real-time programs. These solutions include run-time environments that meet real-time requirements but stay within the limits of the Ada language. Usually these environments provide access to the operating system of a computer through Ada packages -- constructs that define a related group of type definitions, data declarations, functions and procedures.

Concurrency is possible but difficult to achieve in Ada. Users need a way to define tasks that are divided across multiple processors. To do this, programmers must determine the kinds of objects each processor can access, set up communication between processors, and identify the data that must be accessed by each processor. Ada was not designed to meet this complex programming requirement, referred to as the definition of share groups. One solution to this requirement is to provide a mechanism to share data between Ada programs, relieving the user of the low-level implementation that normally occurs with shared data. A pragma defines this package as a shareable data pool.

Ada places operating-system functions in the language. Ada task management is the responsibility of the Ada run-time environment and may or may not involve UNIX process management. The Ada task scheduler sits on top of the UNIX process scheduler which tends to slow things down.

4. Lessons Learned

4.1 Reusability

Customers merely desire fewer defects per system. Reused code is invariably better quality than new code because it has already been "proven." Reusable software building blocks give better quality, a unified user interface across a multiproduct line, and better speed to market.

Reuse can be defined at a broader level than just Ada generics to create "Software-ICs" [3].

We (the royal "we") are close to providing building blocks for a user or applications-provider to easily assemble as if they were a single integrated application from the outset.

Reuse guidelines are needed including definitions, distribution mechanisms, and legal/licensing criteria.

4.2 Rapid Prototyping

Prototype code must be easy to read and analyze because the prototype must support analysis of the intended system and document an initial design. The prototype must be easy to modify because it will be subject to many revisions before the user is satisfied with the requirements as reflected by the prototype's behavior.

It's important to decide in advance how the software development process will be managed and controlled. Unix directory structure and routines provides a shell script hierarchy permitting different segregated prototype instances.

4.3 Multilingual

A multilingual approach provided an initial operating capability: existing user interface software had been written in C but new functions were implemented in Ada.

The use of Ada in a multilingual environment required interprocess communication in which two processes agree on the interchange protocol via files. This approach added complexity but preserved modularity.

Unavailability of Ada-SQL and Ada-POSIX bindings and precocious Ada-X bindings led to use of C as the integration language. A strictly Ada software environment is desired; however, it's still premature to expect Ada to be used exclusively when rapid prototyping since all bindings have not been definitized and disseminated. E.g. pragma interface provided a C interface to POSIX for scandir

Particularly problems can result in multilingual environments word boundaries from the manner that Ada compiler vendors implement type record. Not all vendors contiguously allocate space: embedded nulls and record layout differences have been uncovered when moving the prototype from system to system, Ada compiler to Ada compiler. The Unix od utility has come in handy.

4.4 Ada

Per proper Ada usage, system dependent code was isolated in packages.

TEXT_IO proved exorbitantly expensive when doing text processing. To get better performance (reduce the number of TEXT_IO function calls), an entire message was read into an internal buffer using GET_LINE. Character at a time reading in Ada was not feasible.

ASCII.LF was used to give an end-of-line character similar to C.

UNCHECKED_CONVERSION of ASCII text characters outperformed 'FOS.

4.5 Integrating COTS/NDI Software

There is a significant difference in *cost* between an item uniquely manufactured for a program that will use but small quantities and a *commercially available* equivalent whose standard manufacturing process includes testing to ensure the reliability needed for a tactical or strategic application.

One shortcoming of off-the-shelf software is that it is generic. Users would prefer to make the software adapt more to their needs. This is quite a challenge to us developers, to try and anticipate the many different ways our programs may be used and then build a core architecture that will support that.

COTS/NDI software should be considered an instance of rapid prototyping which provides an initial operating capability. If COTS/NDI software runs in a system-high environment then security considerations diminish.

A transition plan based upon incremental changes to existing fielded software systems can transition (one function at a time) from current source language(s) to Ada. During the transition, the system must be supported by two software environments, the one currently in use and Ada.

This concept is based on an established philosophy of system test engineering: "whenever possible limit the number of unknowns to one." During the transition phase, the system remains fully operational. Only one function at a time is added or modified. Thus only the interfaces to and performance of that function need to be considered during the software updates.

4.6 Interoperability

Full file path names were used to accommodate NFS.

During prototype development SQL was decoupled from client applications to avoid the necessity for licensing agreements when demonstrating the prototype on different hardware. Corporate bureaucracy indeed affects software development since the bureaucracy is not primed to quickly process licensing agreements.

5 Conclusion

Non-embedded C3I systems must be scalable in terms of capacity, storage, and physical size to satisfy site requirements without adversely affecting system functionality and response time capabilities. Proper design and implementation of configurable software components assist these aims.

What Unix gives us is an architecture in which we can integrate multiple architectures. Despite Unix and X Windows, there's still dependencies that an application or systems supplier writes into a system.

Program managers must enforce software discipline and planning.

Prototyping and reuse constitute a more productive approach to software engineering. Using that method, design moves away from a traditional top-down approach to something more iterative, where engineers refine the product until it meets specifications.

Rapid prototyping is particularly effective for ensuring that the requirements accurately reflect the user's real needs, increasing reliability and reducing costly requirements changes.

Rapid prototyping is not the universal panacea for the vexing software development problem. Successful implementation of prototypes is highly individualistic.

Attaining hardware independence (i.e. portable software) requires disciplined application of software engineering techniques.

Some of the management lessons learned regarding Ada include: reusable/portable systems cost more to develop, Ada projects have longer design phases, but shorter coding and integration phases.

Distributed architecture implementation necessitates if not avoidance then judicious use of Ada tasking. If Ada is used like older programming languages, the results are questionable.

Proper use of Ada yields reuse, portability and maintenance benefits. Shortcomings include the lack of a complete inventory of Ada software or Ada projects, lack of a coordinated effort to determine the benefits of Ada and the poor dissemination of completed research on Ada deficiencies.

REFERENCES

1. McCarron, Shane P. Digging Through The Layers. *Unix Review*, July 1988, 62-69.
2. Zuckerman, Susan. Problems With The Multitasking Facilities In The Ada Programming Language. *Defense Communications Engineering Center*, Technical Note No. 16-81, May 1981.
3. Cox, Brad J. Object Oriented Programming: An Evolutionary Approach, Addison-Wesley, 1986.

Tom Dale is a member of the Research Staff at the Unisys McLean Research Center. His background includes work in data communication connectivity, use of Ada in multilingual environments, and development of advanced text handling components. Current research interests include hypertext and data fusion. He may be reached at Unisys Defense Systems, 8201 Greensboro Drive, McLean, VA 22102.

Phone: 703-847-3335
E-mail: dale@mrc.unisys.com
FAX: 703-847-3305

Implementation of Blackboard Systems in Ada

Pamela P. Cook
and

Verlynda S. Dobbs

Department of Computer Science and Engineering
Wright State University
Dayton, Ohio 45435
613-873-2491

Keywords: artificial intelligence in Ada, blackboard systems

Abstract

This paper concentrates on the implementation of blackboard systems in Ada. It describes the blackboard control model of problem solving, a generic approach to the design and implementation of a blackboard control system, and the application to a classic defense problem.

1 Introduction

There is increased interest in implementing artificial intelligence applications in Ada. Efforts in this area encompass expert systems [AdKS6,LaVS6,JI,ZS7], distributed knowledge based systems [BraS6,FraS6], pattern directed processing [RKWS5], semantic networks [SPAS6], reusable heuristic search algorithms [DobSS] and others. This paper concentrates on the implementation of blackboard (BB) systems in Ada. It describes the blackboard control model of problem solving, a generic approach to the design and implementation of a blackboard control system, and the application of the approach to a classic defense problem.

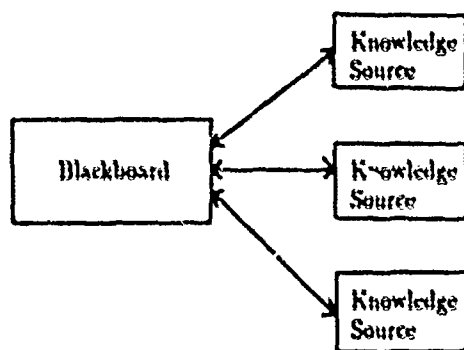
2 Blackboard Model

A problem solving model is a scheme that constructs a solution by organizing reasoning steps and domain knowledge. A model provides a conceptual framework for organizing knowledge and a strategy for applying that knowledge. Examples of problem solving models include forward reasoning models, backward reasoning models, event driven models, model driven models, etc. In a forward reasoning model, the inference steps are applied from an initial state toward a goal. In a backward reasoning model, problem solving begins by reasoning backward from a goal to be achieved toward an initial state.

The blackboard model [NiS6b,NiS6a], which was developed in the 1970's and has undergone very few changes in the last ten years, uses an opportunistic reasoning model. In an opportunistic reasoning model, pieces of knowledge are applied either backward or forward at the most *opportune* time. This model was first abstracted from the Hearsay-II speech understanding system [NiS2] and applied to the design and implementation of the HASP system for ocean surveillance [EHRLRS0]. Many application programs have subsequently been implemented (usually in Lisp) using the blackboard model. These include systems for interpreting electron-density maps, planning errands, understanding military signals, and understanding images.

The basic blackboard model is usually described as consisting of three major components - the knowledge sources, the blackboard data structure, and the control - as shown in the figure on the following page.

*This research is sponsored by the Air Force Office of Scientific Research.



(— data flow)

The domain *knowledge sources* are partitions formed from the total domain knowledge that can be used to solve the problem. These knowledge sources are logically independent and kept separate. The domain *blackboard data structure* is a global data base that holds the problem solving state data. The solution space is organized into one or more application dependent hierarchies. Information at each level of the hierarchy represents partial solutions and is associated with a unique vocabulary that describes the information. The knowledge sources produce changes to the blackboard that lead incrementally to a solution to a problem. Communication and interaction among knowledge sources take place solely through the blackboard. There is no pre-determined flow of control. The knowledge sources respond opportunistically to changes in the blackboard. The knowledge sources transform information on one level of the hierarchy into information on the same or other levels using algorithmic procedures or heuristic rules that generate actual or hypothetical transformations. Which knowledge source to apply is determined dynamically, one step at a time, resulting in the incremental generation of partial solutions. The choice of a knowledge source is based on the solution state and on the existence of knowledge sources capable of improving the current state of the solution.

There are at least two different approaches to handling the control. The first has control residing in a set of procedures which monitor the changes on the domain blackboard and trigger appropriate knowledge sources to improve the solution state. In the second approach, control is achieved by placing the strategy on a control blackboard. The decision then as to what to do next is made by control knowledge sources using the control blackboard where data describing the state of the current strategy exists. Strategies can be enabled on the control blackboard to reflect the current state of the problem solution. This second approach is referred to as a blackboard control architecture [HRS5].

The blackboard architecture approach improves on traditional expert systems for solving ill-structured problems in the following ways. First, the blackboard approach requires no a priori determined reasoning path. Because ill-structured problems often do not have a pre-determined decision path to a solution, the selection of what to do next must be made while the problem is being solved. The capability to do this is provided in blackboard systems by the incremental and opportunistic problem solving approach. Second, vague information and knowledge, which characterize ill-structured problems, need to be made concrete in the process of finding a solution to the problem. The blackboard model is an excellent tool for this knowledge engineering activity. During the initial interactions with an expert, a knowledge engineer tries to find an appropriate conceptual model for the task while trying to understand the domain and the nature of the task. The blackboard approach aids in the problem formulation because it provides some organizational principles that are both powerful and flexible. The blackboard approach is also an excellent tool for exploratory programming, a useful technique for developing solutions to complex and ill-structured problems.

Although blackboard systems are useful for many complex, ill-structured problems, they are generally expensive to build and to use. Therefore, the blackboard approach should not be used when lower cost methods are sufficient. A problem which exhibits some combination of the following characteristics is a good candidate for the blackboard approach:

1. a large solution space
2. noisy and unreliable information
3. a variety of input data and a need to integrate diverse information
4. the need for many independent or semi-independent pieces of knowledge to cooperate in forming a solution
5. the need to use multiple reasoning methods or lines of reasoning
6. the need for an evolutionary solution

A proposed application should be carefully analyzed before a decision to implement a blackboard system is made.

3 Application of BB Systems

The following sections describe the domain analysis, functional requirements, design, and implementation of the prototype system for a classic defense problem.

3.1 Domain Analysis

Functions of a classic defense system include sensing of the environment; interpretation of conflicting, incomplete or corrupted sensory data; integration of sensory data; overall situation assessment; planning and real-time decisions for mission and survival achievement; and control of weapons system functions. These activities, which are currently performed by the crew, would ideally be provided by a ground and/or onboard computer system. The system would be able to function in an environment containing vast amounts of raw data where the data may be unintentionally corrupted via natural phenomena or intentionally corrupted via friendly or enemy jamming and deception missions.

Electronic warfare (EW) processing spans many different disciplines (passive radio frequency (RF) and infrared (IR) sensor interpretation, active RF and IR countermeasures techniques, etc.) where the sensors operate in different environments with different requirements for information extraction, interpretation and reaction. The EW system must respond to a dense and dynamic environment. A priori information is used to distinguish threats to the aircraft from non-threats and also contributes to the system response and resource allocation strategy. A priori information often does not represent the true state of the environment because of environment noise, intentional deceptive emissions, jamming, etc.

From this description it is easy to see that EW exhibits several of the criteria that make a good candidate for a blackboard system. It has a large solution space that includes knowledge concerning static threats, limited resources, passive sensors, terrain data, platform data, active countermeasures, goals to be accomplished, etc. Some of the data is noisy and unreliable. There is a variety of input data and a need to integrate diverse information. There is also the need for many independent pieces of knowledge to cooperate in forming a solution. The different types of knowledge require different vocabularies and different lines of reasoning. The solution evolves as new pieces of information are sensed, input and derived from existing data.

One EW defense problem is the monitoring of a hostile environment by a moving platform for the purpose of determining the platform's best path through the environment. Emissions are detected and locations and types of emitters are determined. A map is produced which represents a snapshot of the current knowledge about the environment. This map can then be used as input to a program that will determine the best path for the platform through this environment [DDLS]. This monitoring problem was chosen for the prototype of the blackboard control system in Ada. The domain blackboard will contain the current situation board with respect to the environment while the control blackboard will contain data that indicates strategies and reasonings to be employed.

3.2 Functional Requirements

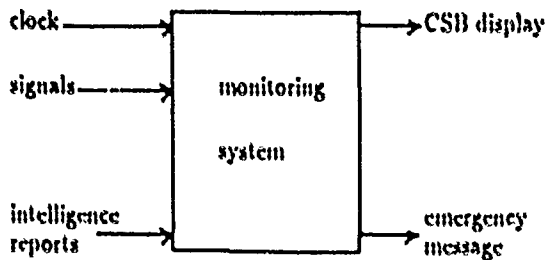
The functional requirements for the prototype include:

1. Process batches of input data at regular intervals of 4 time units. Input data can be either signal data or intelligence reports. Signal data consists of three emitter characteristics - frequency, pulse width and pulse repetition frequency; a location represented by a position on a grid; and a signal detection time. Intelligence report data, which is for threats only, consists of the location and sighting report time.
2. Determine type of emitter from characteristics of each set of signal data.
3. Identify each emitter as threat or nonthreat based on the capabilities of the emitter type.
4. Post to the current situation board (CSB) the locations of known threats and nonthreats and the time of sighting.
5. Eliminate duplicate sightings but keep history information for each location.
6. Keep a list of known friendly locations to determine whether any emitter identified as a threat is a known friend.
7. Output CSB at regular intervals of 10 time units, including location of sighting, threat or nonthreat, history information, and confidence.
8. Output SOS message immediately when possible threat is detected.

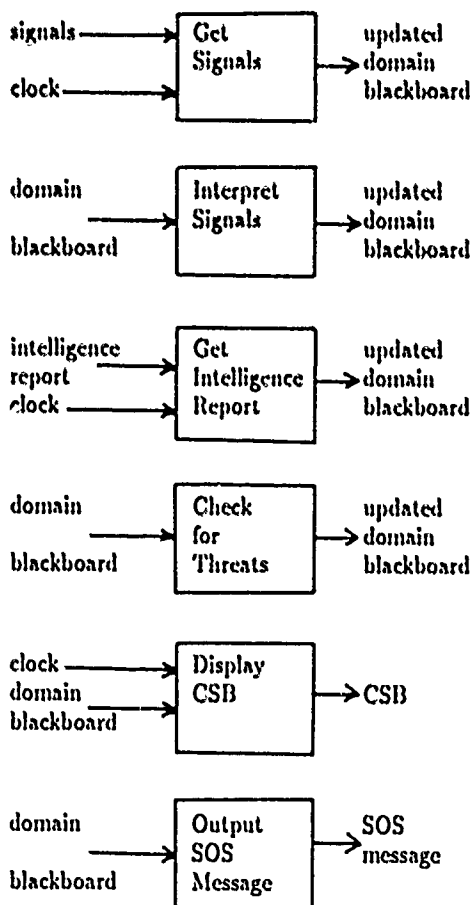
3.3 Design

Two different design methods were considered for the system - object oriented and functional. The choice of functional design was based on the fact that blackboard systems and the blackboard control architecture have been described functionally [HRS5]. As the system is decomposed, the functional design approach identifies major tasks to be performed. Subprograms (program units) become the building blocks of a functional design. The system is described in terms of the functions that process the data. Functions receive input, transform input, and produce output. The functional requirements define what needs to be done. The functional design includes the requirements as well as the data flow. Pictured below are the various levels of a functional design.

At the highest level, a block diagram of the system showing inputs and outputs would look like this:



The major function of the monitoring system can then be broken down into a set of functions, each with inputs and outputs.



3.4 Implementation

The blackboard control architecture model defines two blackboards, one for the domain and one for control. A set of data structures and a set of knowledge sources are associated with each blackboard. The data structure for each blackboard has its own hierarchy. Each of these components is defined below for the monitoring problem.

3.4.1 Domain Blackboard

The hierarchy for the domain blackboard contains three levels of abstraction: signals, emitters and sites. The domain knowledge sources are listed below:

- **KS0.** Initializes table of locations of known friends on domain blackboard.
- **KS1.** Reads input data, creates signal node on signal level of domain blackboard or creates expectation on site level of domain blackboard.
- **KS2.** Generates emitter node on emitter level of domain blackboard based on characteristics of signal node.
- **KS3.** Generates site node on site level of domain blackboard based on characteristics of emitter node and other information on domain blackboard.
- **KS5.** Handles duplicate emitter nodes on emitter level of domain blackboard, updating history information.
- **KS6.** Initiates termination of system.
- **KS7.** Prints SOS message when possible threat is sighted.
- **KS8.** Outputs CSB and associated information.

Execution of the knowledge sources reflects changes in the environment. Both forward and backward reasoning are used on the domain blackboard. Forward reasoning takes signal input data, creates emitter types from the signal data, and generates threat or nonthreat sites from the emitter types. Backward or model driven reasoning occurs when intelligence reports indicate a threat at a specific location at a specific time. The site can then be verified based on the intelligence report. Discrepancies at this level result from noisy or unreliable data.

3.4.2 Control Blackboard

The control blackboard contains three levels of abstraction that define the strategy used for handling events that occur in the system. These levels are policy, tactic and step. Policy, located on the highest level, determines which blackboard is effected - domain or control. Tactic, on the next level, specifies the category of event to process. The lowest level of abstraction, step, differentiates between the two low level event types.

The control knowledge sources toggle between possible values on each level of abstraction to implement the chosen strategy.

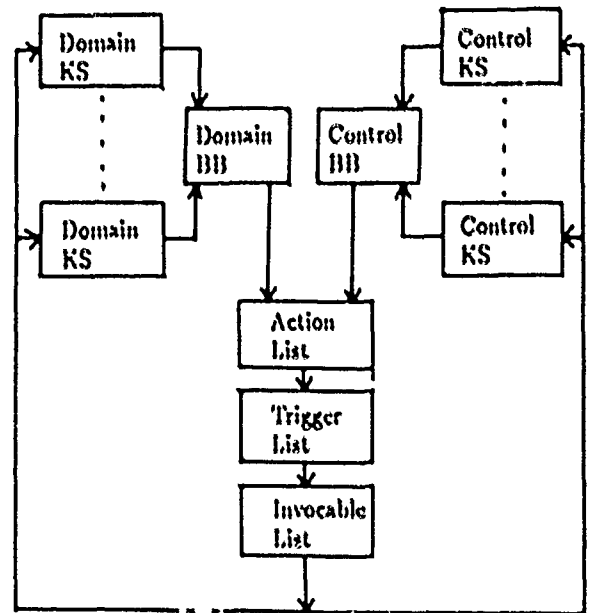
- **KS10.** Initializes the system with problem description from user.
- **KS11.** Initializes control blackboard by setting up policy, tactic and step level for policy being used.
- **KS12.** Generates event to change policy if policy is not "control".
- **KS13.** Toggles tactic setting.
- **KS14.** Toggles step setting.
- **KS20.** Toggles policy setting.

Three additional domain-independent procedures - update trigger list, choose KSAR, and execute KSAR - drive an iterative control strategy that operates around three event lists - action, trigger and invocable. Update trigger list moves KSARs from the action list to the trigger list. Choose KSAR matches conditions of control and the environment with the KSAR and moves the KSAR to the invocable list if the conditions are met. Execute KSAR executes the knowledge source of the first KSAR on the invocable list.

3.4.3 Operation

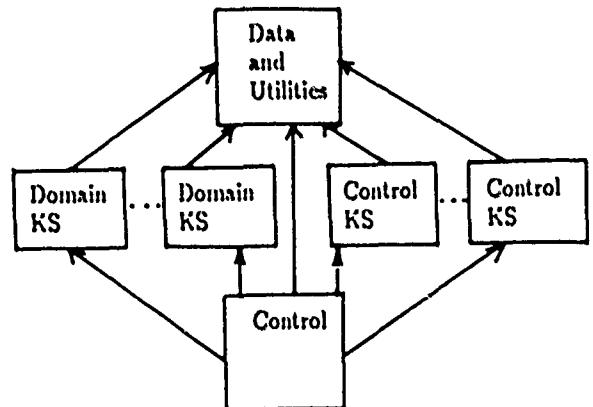
Updating the domain blackboard creates events that reflect the updated environment. These events are recorded in knowledge source activation records (KSARs). A KSAR contains information on the triggering cycle, triggering event, precondition values, condition values, trigger weight, knowledge source importance, event type, rating and priority. The KSAR is initially placed on the action event list. The action event list contains all KSARs generated by the previous cycle's execution. As KSARs on the action event list are moved to the trigger event list, the events are given a rating based on the current state of the environment and the importance of the associated knowledge source. The KSARs are placed on the trigger event list in descending order by rating points. KSARs are moved to the invocable event list based on the current control strategy. The knowledge source of the first KSAR on the invocable list is executed. If no knowledge source is triggered on a particular cycle, an event is created that will enable a change in policy, tactic or step level on the control blackboard.

The action, trigger and invocable event lists contain both domain and control events. The policy, tactic and step settings determine which KSARs get moved to the invocable list. Only one event activates a knowledge source (KS) each cycle - the event that is first on the invocable list. The execution of a knowledge source on a cycle generates new KSARs on the action list. A general diagram of the system is shown below.



4 Use of Ada

The monitoring system was developed in Vax Ada on a Vax 11/780 running the VMS operating system. The use of Ada for this prototype presented no major problems. The resulting system has the structure shown below. (The arrows indicate required visibility.)



Several of Ada's features greatly facilitated the implementation. The package concept facilitates data encapsulation, modularity, and locality. Operations on a particular object, for example an event, are contained in a package with the implementation details hidden from users of the package. Changes in the implementation will not effect the control structure or the object's interaction with other objects.

The modularity feature of the package encourages reusability as basic operations can be reused and not reprogrammed. Packages were used to contain:

- blackboards with associated procedures (data and utilities package)
- procedures to execute cycle for control (control)
- knowledge sources (KSs)

The data and utilities package contains all global structures and utilities. The control and domain blackboards are defined in this package along with procedures that delete information from the domain blackboard as well as create, add and delete nodes from the action, trigger, and invocable event lists. Much of this package is reusable since it manipulates the structures for a blackboard control architecture.

Each knowledge source is contained in a separate package. New knowledge sources can be added and existing ones changed without recompiling the entire system. (Instead of packages, knowledge sources could be contained in separately compilable procedure modules.) Keeping these knowledge sources separate will also facilitate the introduction of concurrency into the system using the tasking feature. More than one knowledge source could then be executed in a cycle.

Other Ada features were also useful in the development of this system. Variant records were used to represent blackboard nodes since the nodes on the different levels of the domain blackboard contained different information fields. Using variant records made it possible to manipulate the nodes on all levels with the same procedures.

The one disadvantage that we encountered resulted from changes in the structure or information that was on one of the blackboards. Any changes in the data and utilities package made it necessary to recompile all other units. Changing the tables from static to dynamic and initializing them from files would eliminate many of the changes that were required.

5 Conclusions

A prototype monitoring system based on the blackboard control model of problem solving was successfully implemented in Ada. Blackboard systems exhibit several characteristics that parallel features of Ada. The independent knowledge sources can be developed as program units. Packages can encapsulate operations on specific types of data. The tasking mechanism can provide the dynamic property required by the control mechanism. The use of Ada tasks will allow a natural movement to multiprocessor systems. These features, along with the capability to incrementally build a solution, make Ada a good choice for a blackboard system.

The incorporation of AI techniques into solutions of real world problems has been hampered by the use of speciality languages for their implementation. Ada provides the vehicle for moving these techniques out of the research laboratories. Successful integration of artificial intelligence techniques into Ada will create adaptable software packages that are highly suitable for solving real world problems.

References

- [Adk86] M. Adkins. Flexible data and control structures in ada. In *2nd Annual Conference on Artificial Intelligence and Ada*, 1986.
- [Bra86] D. Brauer. Ada and knowledge-based systems: A prototype combining the best of both worlds. In *First International Conference on Ada Programming Language Applications for the NASA Space Station*, 1986.
- [DDL88] V. Dobbs, H. Davis, and C. Lizza. An application of heuristic search techniques to the problem of flight path generation in a military hostile environment. In *1st International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 1988.
- [Dob88] V. Dobbs. Reusable ada modules for artificial intelligence applications. In *6th National Conference on Ada Technology*, 1988.
- [EHRLRS0] L. Erman, F. Hayes-Roth, V. Lesser, and D. Reddy. The hearst-II speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12:213-251, June 1980.
- [Fra86] M. Frank. Using ada to implement the operations management system as a community of experts. In *First International Conference on Ada Programming Language Applications for the NASA Space Station*, 1986.
- [HRS5] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26:251-321, 1985.
- [JLZ87] A. Jaworski, D. LaVallee, and D. Zoch. A lisp-ada connection for expert system development. In *3rd Annual Conference on Artificial Intelligence and Ada*, 1987.
- [LaV86] D. LaVallee. An ada inference engine for expert systems. In *First International Conference on Ada Programming Language Applications for the NASA Space Station*, 1986.
- [Nii82] H.P. Nii. Signal-to-symbol transformation: Hasp/siap case study. *AI Magazine*, Spring 1982.
- [Nii86a] H.P. Nii. Blackboard application systems and a knowledge engineering perspective. *AI Magazine*, 7(4), August 1986.
- [Nii86b] H.P. Nii. The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(3), Summer 1986.

- [RKWS5] L. Recker, J. Kreuter, and K. Wauchope. Artificial intelligence in ada: Pattern-directed processing. Technical Report AFHRL-TR-12, AFHRL, May 1985.
- [SPAS6] D. Scheidt, D. Preston, and M. Armstrong. Implementing semantic networks in ada. In *2nd Annual Conference on Artificial Intelligence and Ada*, 1986.



Pamela Cook received the M.S. degree in computer science from the University of Dayton in 1979 and has been pursuing a Ph.D. in computer science at Wright State University. She has held computer scientist positions with Ford Motor Credit Company, General Motors, and Federal-Mogul.



Verlynda Dobbs received her Ph.D. in computer science from The Ohio State University in 1985. Her research interests are in the areas of software engineering, artificial intelligence, and Ada for artificial intelligence. Dr. Dobbs is currently on the faculty of the Department of Computer Science and Engineering at Wright State University, Dayton, Ohio 45435.

The AN/TSC-99 Redesign Effort -
An Experience in Software Engineering with Ada

E. Peter Gunderson

David A. Vaughn

Gordon E. Bostic II

Telos Federal Systems

Telos Federal Systems

Telos Federal Systems

ABSTRACT

This paper examines the first attempt of a small development team with no experience in Ada or Object-Oriented Design, to design a real-time system utilizing Ada. The developed software controls radio communications equipment. It will replace the existing software used to send and receive digital messages and allocate equipment on a timed schedule. The success of this development proves the viability of extending the useful life of aging systems by replacing software and upgrading processors at a much lower cost than developing a new system. This paper relates to both new users of Ada, who will possibly encounter many of the same problems; and to the experienced software designer who may avoid similar problems.

INTRODUCTION

The U.S. Army Communications-Electronics Command (CECOM) Center for Software Engineering (CSE), Fort Monmouth, NJ, tasked Telos Federal Systems to conduct a feasibility study involving the redesign and rehost of AN/TSC-99 software. The objectives of this task were to perform the following:

a. Provide a working understanding of Ada as it applies to equipment control and real-time scheduling functions

b. Provide a test bed model for requirements analysis of system

interfaces, especially operator key-board entry and displays

c. Provide hands-on software engineering training using Ada to explore timing and responsiveness, memory, sizing, and multi-tasking designs

d. Evaluate program support environment tools and compilers through operation and benchmarks

e. Prove the viability of using Ada to control the AN/TSC-99.

1.0 THE SYSTEM

The AN/TSC-99 is a computer controlled, High Frequency (HF) Radio, Burst Communications System. It is contained in two shelters; the Receiver Group Subsystem (RGS), and the Transmitter Group Subsystem (TGS). These shelters can be separated by up to five miles. The RGS processor controls four HF transmitters, five receivers, and one satellite transceiver. Normally, the system is controlled by the RGS processor. In the event of RGS system failure, the TGS can operate at reduced capacity in the off-line mode. RGS software also controls other devices such as paper-tape punches and readers. Device interfaces are serial using RS-232 protocol. The current fielded version of the AN/TSC-99 is written in ROLM assembly language and targeted to the military computer family equipment, which includes the ROLM 1602B processor, ROLM 2150 I/O chassis, and a DATARAM disk emulator. The system configuration is shown in Figure 1.

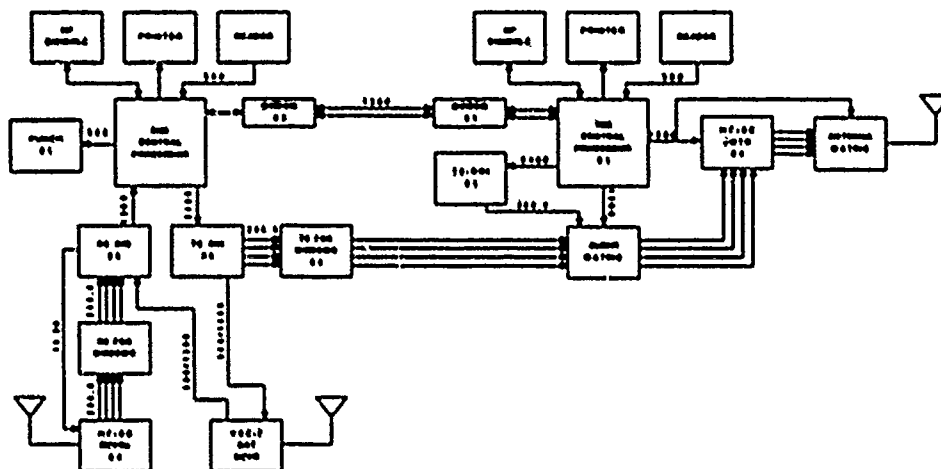


FIGURE 1. SYSTEM CONFIGURATION

1.1 THE SOFTWARE PROBLEM

The AN/TSC-99 could no longer support changes in operational requirements, primarily due to a lack of memory. Developing a paging scheme was considered but deemed unacceptable because of the cost of development and the expected degradation in performance. Other considerations supporting a redesign effort included the elimination of obsolete peripherals, incorporation of modern communications security equipment, and adaptation of software to different hardware configurations.

1.2 THE TARGET PROCESSOR

TELOS intended to provide a rapid prototype of a functionally equivalent system written in Ada and capable of running on a modern high-speed processor. Zenith 2248s were chosen as both host and target processors for the prototype. While there were drawbacks to this configuration, this solution was chosen because 2248s were already in inventory and we wanted to keep development costs to a minimum. The portability of Ada would allow porting to

another processor with minimum effort if required. Meridian's 80286 Ada compiler was selected because it could run on a standard PC and was validated by the Ada Joint Program Office (AJPO). The hardware configuration of the prototype is shown in Figure 2.

2.0 DESIGN GOALS

We entered into this Feasibility Study with three separate design objectives: (1) to develop software that would lend itself to easy upgrade; (2) to maintain the same basic user interfaces, using an extension of the existing AN/TSC-99 Users Guide as a functional specification; and (3) to make the code portable and reusable to the greatest extent possible.

3.0 PROJECT ORGANIZATION

The project was organized under a System Architect who performed the duties of the Chief Programmer. The working organization is shown in Figure 3. Two software engineers worked under the System Architect, one concentrating in peripheral

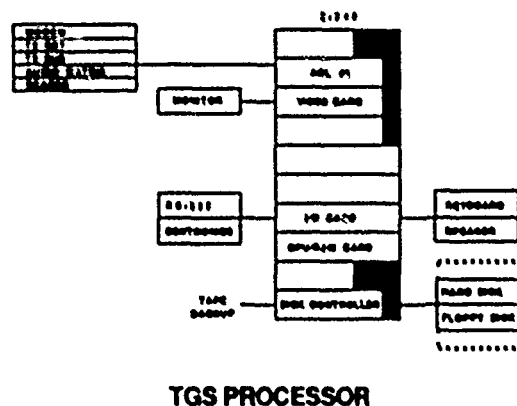
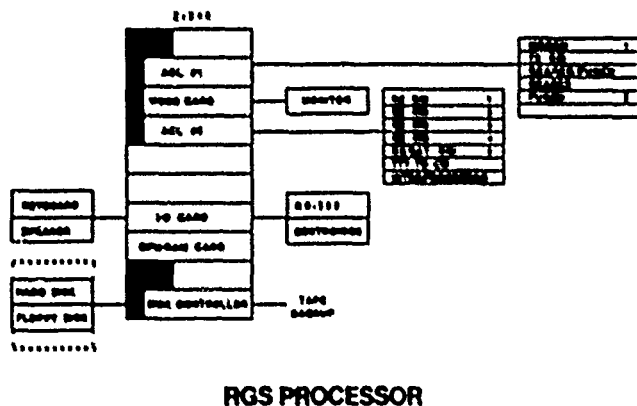


FIGURE 2. AN/TSC-99 PROTOTYPE CONFIGURATION WITH ACL CARDS

interfaces programmed in assembly language. The second devoted to the overall design and integration of the Ada programs.

3.1 PROJECT DEVELOPMENT PLAN

Development followed the planned schedule highlights:

a. We estimated one year would be the required to complete the project as the requirements were clearly understood, target equipment was in place and, with the exception of the new computers, fully integrated.

b. Four engineers were to be dedicated to the design of the proof of concept prototype. If the prototype proved successful the staff would be expanded to six engineers.

c. We would train on the job, teaching ourselves both Ada and Object-Oriented Design techniques.

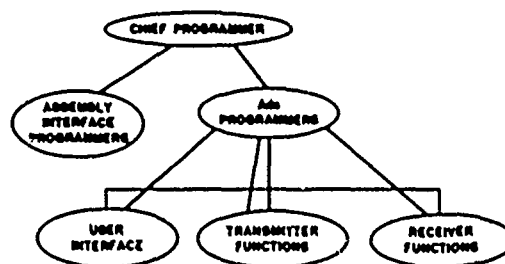


FIGURE 3. TEAM ORGANIZATION

3.2 FACILITIES AND SUPPORT SOFTWARE

One office was to be used as a software development lab. This office would contain three Z248s and one printer. A separate office would be used as a test facility. It would contain two Z248s and three IBM XTs. The Z248s would be used as target processors (one RGS and one TGS). The XTs were to be used to emulate equipment in the AN/TSC-99. Test facility layout is shown in Figure 4. Much time was spent on the development and maintenance of our facilities. Software was written to emulate radio interface cards and paper-tape readers, create message data, and monitor data lines. Additional software was written to automate compiling and linking of Ada programs. C, Assembly, and Ada were used in writing this software. In all, 26 support programs were written. The development spanned four versions of the Ada compiler. Installing and recompiling with each new release was time consuming as complete compatibility between versions was not always maintained.

3.3 COMMUNICATION

Most communication within the team was verbal. Design changes were discussed and agreed to during design reviews. The team was managed by a chief programmer who coordinated all activities and facilitated direction and control. This approach was chosen to expedite development. Documentation was kept to a minimum although skeletons of most DOD-STD-2167 required documents were generated, including a software requirements specification and top-level design document.

3.4 CONFIGURATION MANAGEMENT/QUALITY ASSURANCE

Configuration Management (CM) was applied to two Ada baselines, one TGS and one RGS. Once developed, each was controlled independently. The moving baselines were continually maintained and completely updated after each integration test. PC-based metrics software was used to monitor program coding techniques.

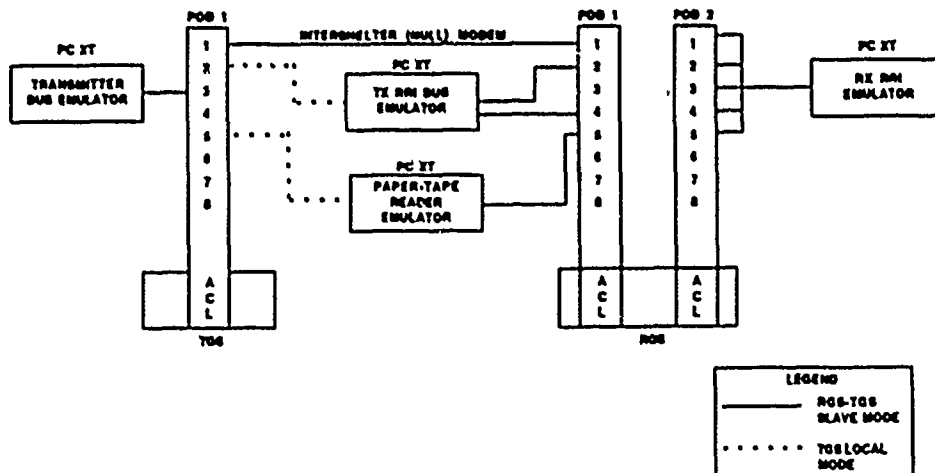


FIGURE 4. IN-HOUSE SOFTWARE TEST ENVIRONMENT

3.5 INTEGRATION AND TESTING

Existing AN/TSC-99 test procedures were used for system level testing. Unit-level testing was performed by developing engineers. Integration testing provided full string transmission of messages through the system.

4.0 APPROACH

4.1 PERIPHERAL CONTROL

Knowing the limitations of implementing a real-time system running under MS-DOS, our first consideration was how to handle interrupts from 16 serial ports. We decided to use distributed processing, purchasing off-the-shelf Advanced Communication Link (ACL) boards manufactured by Stargate Technology. Each ACL contained eight user-configurable serial ports. Hardware interrupts were handled by an on-board Intel 8088 microprocessor. Information was passed to the 2248s through a 32k window of dual-port memory. Assembly language routines were used to service the ACL board. These routines updated a service table.

4.2 SOFTWARE

Ada-tasking was used to monitor the service table, instantiating other tasks as required. The existing system-level specification was used to determine requirements of the software. We coded a few functionally equivalent procedures in Ada in an attempt to get some rough sizing data. We estimated the Ada implementation would be about two to three times as large as the existing Assembly language implementation.

4.3 TIMING

All critical timing (i.e., hardware interrupts) was handled by existing embedded firmware, 80286 assembly language routines and the ACL boards. Ada tasks were used where timing was less critical, such as allocating equipment and scheduling outgoing message traffic. Equipment allocation began five minutes prior to message transmission. A one minute window was allowed for transmission. The service table used to activate these tasks was monitored once every second.

5.0 DESIGN IMPLEMENTATION

We decided to first design and prototype a skeleton of the TGS side of the system. The idea was to prove the viability of our concept. This model would simply load, schedule, and transmit a digital message over HF radio. The message would be captured in the KCS using existing software. If successful, the paradigm would be expanded.

5.1 TOP LEVEL

A combination of Structured Analysis and Design and Object-Oriented Design techniques were used. The static structure of the system was developed and interpackage dependencies were loosely defined. Whenever possible dependencies were incorporated into the package bodies. This was done to reduce compile times as new dependencies were identified. Data structures were written for peripheral interfaces and displays.

Rapid prototyping was used to develop the framework of the system. The top-level design was revised. Writing of package specifications was begun.

5.2 DETAILED LEVEL

The paradigm was further expanded. The unit-level interfaces were fully defined. Package specifications were completed. At this point, we found the top-level design often required revision.

5.3 CODING

After testing the proof of concept model, implementation proceeded from two directions. The low-level (Assembly) modules were tackled by one team. The high-level (Ada) procedures were coded top-down by another. Engineers were provided with the specification for each procedure. The procedure was then coded and tested at the unit level. When testing was completed, the procedure was integrated into the package. As packages were completed, integration testing was done at the interpackage level. Emulation software was written for several devices, in order to lessen the number of trips required to Tobyhanna Army Depot, Tobyhanna, Pennsylvania, where a complete

AN/TSC-99 test bed is located. Full system testing was accomplished using the system at Tobyhanna.

6.0 ACCOMPLISHMENTS

The resulting executable system is approximately 2.5 times the size of the old system. We still have a 50% memory reserve in the RCS and 65% in the TGS. Source code is approximately 28,000 lines. The old system contained approximately 204,000. Much improvement has been realized in maintainability. System response is comparable to the old system. We feel we can improve this slightly under the current operating system. A multi-tasking real-time operating system would probably provide substantial improvement. About 40% of the code was reused between the TGS and RCS programs. No conclusions can be reached as to portability as we have not attempted to port to another processor. Productivity averaged approximately 28 lines of code each day per engineer. This was affected negatively by the long learning curve and positively by the high reusability of code within the system.

7.0 LESSONS LEARNED

As a result of this Feasibility Study, we would like to suggest the following:

a. Users should fully understand the Design Methodology being used.

Understanding Object-Oriented Design techniques was one of our most difficult tasks. Few books on the subject were available and many questions went unanswered. Most of our knowledge came from papers written on the subject. The methods used generally followed structured analysis and design, however, Top-Level Design of the Ada programs attempted to apply the Object-Oriented Design for package construction. This gave us good data flows for procedures and minimal package specifications dependencies, but very large data structures as well.

b. Use rapid prototyping for evaluating design approaches and requirements analysis but stop short of full implementation.

We found that changes to the Top-level design were being driven by detail implementation. The problem was caused, in part, by never getting out of the rapid prototyping mode. If automated design tools had been available, this would have been less likely.

c. Follow a methodology and enforce coding standards.

While our source listings are easy to read, their format varies throughout the system. Using metrics to monitor development contributed to the maintainability of code, and helped us to identify code that was too complex or lacking in comments. As a result, procedures were kept simple and contained, on average, 40% comments. The point of maintainability was brought home by observing how quickly new members assigned to the team became productive. TELOS has now instituted formal methodology for use in developing systems implemented in Ada.

d. Emphasize reusability.

Reusability saved time and effort. Since the TGS requirements were essentially a subset of the RCS functions, much reusability came naturally. Handling of incoming and outgoing messages in the RCS shared many common qualities, which allowed reusability by design. Having spent more time on identifying reusable functions could have eliminated duplicated code.

e. Share files/packages during development.

We initially believed that we could develop code in our own directories and port the finished product to a package in a central directory. We had hoped that the use of a "separate" clause would allow this, but the compiler failed to support this feature after 100 separate procedures had been completed. With a staff of six engineers, this proved to be an inconvenience. Had the system been larger, we would have developed major problems. Proper utilization of Configuration Management during development stages would have helped to avoid the headaches caused.

f. Select a mature compiler.

Our compiler was the cause of many problems. In early releases many features did not work correctly. As the compiler matured most problems were corrected. In the current release, Version 2.2, the only major deficiency we've found is the corruption of heap storage when using string concatenation. This problem has been overcome by using slices to construct strings.

The problems we encountered are not unique to Meridian. Ada is the most complex language ever developed and all Ada compilers have suffered the same growing pains. In fact, Meridian's compiler has many valued features. It interfaces easily with assembly and C code. Several support packages such as the DOS environment and utilities packages are also available. We feel Meridian's compiler represents a good value for the money.

ACKNOWLEDGMENTS

We would like to thank Mr. Walter Lucas of CECOM CSE for giving us the opportunity to attempt this study. Without his support and faith, this project would not have been possible. We would also like to thank the other members of our team who all worked so hard to bring this design to life, as well as the many people at TELOS from whose experience we drew.

ABOUT THE AUTHORS

Mr. Gunderson is a Senior Software Engineer with Telos Federal Systems. He has worked as an instructor, consultant, and engineer in the computer industry. He has design experience in business systems, automatic test equipment, and systems software, and has been working with military systems for the past ten years.

Mr. Gunderson earned his AS degree in General Education from Ocean County College, Toms River, New Jersey, and is working toward his BS degree in Computer Science at Thomas Edison College, Trenton, New Jersey.

Mr. Vaughn is currently a Scientific Programmer with International Telephone and Telegraph. He has worked in both hardware and software engineering, and has eight years of experience in military systems.

Mr. Vaughn is working toward his BS degree in Computer Science at Thomas Edison College, Trenton, New Jersey.

Mr. Nostic is currently a Systems Software Developer with Computer Associates International. He has experience in both Real-Time and Operating System Software Design.

Mr. Nostic earned his BA degree in Social Science from James Madison University, Harrisburg, Virginia, and his Masters degree in Computer Science from Fairleigh Dickinson University, Rutherford, New Jersey.

A PRACTICAL APPROACH TO METHODOLOGIES, ADA AND DOD-STD-2167A

Karen S. Ellison, Ph.D. and William J. Goulet

Jet Propulsion Laboratory, California Institute of Technology
Pasadena, California 91109

SUMMARY

This paper presents a complete methodology for the design phase of a large real-time software project. It assumes the project is being developed under DOD-STD-2167A¹ and will be implemented in Ada. In DOD-STD-2167A terms, both the preliminary design and detailed design phases are covered. Because the methodology presented in this paper incorporates ideas from most of the methodologies currently in vogue, it is called the Hybrid Methodology. The Hybrid Methodology has two purposes: to provide an outline of the engineering steps required by a team of software developers to design a real-time system, and to relate the engineering process to the Software Design Document (SDD)² required by DOD-STD-2167A.

Introduction

There has been much written on software engineering methodologies, especially with the advent of Ada. Several are specifically geared to developing software in Ada^{3,4}. Several address developing real-time systems^{5,6}. Some deal with the transition from requirements definition to design^{7,8}. None relate the design process to the DOD-STD-2167A products and phases. In fact, Donald Firesmith gives lectures on Ada project management⁹ telling managers that Ada and DOD-STD-2167A do not fit.

The Hybrid Methodology

The Hybrid Methodology provides an approach to developing the Software Design Document as a natural by-product of the engineering process. It takes advantage of the expressiveness of Ada to present the design information so that it evolves naturally into an implementation. All aspects of designing software for a large real-time system are taken into account, including early design of the implementation framework of the system (the software architecture concepts), support of discrete event simulation of the system, and traceability to requirements.

Design Phases

The Hybrid Methodology is presented in two parts: preliminary design and detailed design. The paper assumes that the requirements analysis phase has been completed and that there is a Software Requirements Specification (SRS)¹⁰, and a system data flow diagram.

The first part of preliminary design provides traceability to the SRS and transitions from a requirements view of the system to an implementation view. Both the preliminary design and detailed design phases are subdivided into steps which relate to the engineering process. Each step culminates in documenting the design at that point in appropriate sections of the Software Design Document. The Software Design Document is therefore incrementally developed and reviewed as the engineering is done and not in a rush shortly before delivery.

The Hybrid Methodology is presented assuming a system consisting of a single Computer Software Configuration Item (CSCI). However, it can also be applied to a system consisting of multiple CSCIs. Usually the partitioning of a system into multiple CSCIs takes place before the SRSs are written. The CSCIs are usually defined on functional (and possibly hardware) boundaries. The Hybrid Methodology in this case would begin by writing the architecture concepts paper described below for the system as a whole. Section 3.1 of the SDD for each CSCI would then be described in terms of the concepts in the architecture concepts paper.

Methodology for Preliminary Design

Preliminary design consists of four steps. Each step refines the design produced at the previous step. Products are developed which serve to document the complete design at each step and to serve as a vehicle for informal review. In addition, some of the products serve as engineering workpapers which will be developed into deliverable documents. The four steps are:

1. Identify top-level CSCs
2. Identify sub-level CSCs
3. Produce formal documentation for each sub-level CSC
4. Describe abstract algorithm of each sub-level CSC

The description of each step starts with an overview of the step. If there are concepts which help in understanding the products to be developed during the step, they are presented following the overview. The products of the step are then described. Finally a list of deliverables or sections of deliverables which are completed by this step is provided.

Step 1 - Identify top-level CSCs.

Top-level CSCs are the major functional entities of the system. They are derived by analyzing the system level dataflow. For a real-time system, one starts at the external interfaces and works in, identifying the major processing done to handle each interface. There should be no more than 5-7 top-level CSCs for understandability.

After the top-level CSCs are identified, the software architecture framework of the system is designed. This will be documented in a Software Architecture Concepts paper which will be preserved in the CSCI Software Development File (SDF).

Overview on Software Architecture Concepts.

Before individual teams can begin designing their top-level CSCs, they must understand how their piece fits into the system from an implementation point of view. This requires a top-down design of the mechanisms which tie the system together. These mechanisms can then be implemented in such a way as to isolate the "applications" from the underlying operating system and hardware. This framework also provides the top down design of mechanisms which control and detect errors in the system. Once these mechanisms have been engineered on a system level, there will be derived requirements levied on the software (and possibly hardware) in order to implement those mechanisms.

Software Architecture Topics. The software architecture of the system is described for two areas. The first is Monitor and Control. Concepts must be developed top down for how the system is to be controlled, and what is to be monitored. The focus should be on thinking in terms of keeping the operator informed, and in terms of what the operator must control. The second area is System Services. These services are used by the "applications" of the system to communicate with each other, to access common data, and perform functions in a system-defined way. The intent is to hide the specifics of the operating system from the rest of the software, and to provide a common framework for the software.

Monitor and Control. Monitor and Control must take a system view. Concepts must be developed which relate how individual components of the software work together to perform startup, shutdown, fault detection and recovery.

a. Startup/startover/shutdown

Concepts must be developed for starting the system, restarting in case of a crash, and shutting down the system. What applications can be restarted? What state data must be maintained? What mechanisms will be used for recovering data before the system restarts? How will

the system be terminated gracefully? Will functions be able to be started/terminated individually? Will functions require entries which can be called by Monitor and Control to start up or terminate?

b. Fault detection/recovery

What faults must the system detect? From what faults must it recovery? What are the mechanisms by which the faults will be detected? How will errors be reported to the operator? What actions will the operator be able to take to recovery from faults?

c. Miscellaneous

What are the possible configurations of the system? How will the system be reconfigured? Will the system run in degraded modes? How will this work?

System services. System services are needed to provide common ways of passing messages, accessing data, logging, and performing algorithms common to several applications of the system.

a. Message passing

Application functional entities which may be implemented on different processors, or which may move from one processor to another due to a reconfiguration, or which should be decoupled for flexibility in configuring the system will communicate via a system service. They will send a message to a functional address. They will receive messages by identifying themselves as a particular functional address. The list of functional addresses must be established. The mechanisms for routing data using the functional address concept and for reconfiguration must be developed.

b. Database

The operational needs for data must be determined. This includes adaptation data as well as data generated during operation of the system. Where the data will reside, how it will be distributed, mechanisms for maintaining consistency and validity of the data must be determined. Mechanisms for updating, reading the data must be designed.

c. Logging

Requirements for logging data must be

determined. What data should be logged, how will it be logged? On a distributed system one must decide if logging will be done on one processor or on each processor, etc. Control of logging must be designed. Will categories of data be enabled/disabled for logging? What are the categories for this system? What will the mechanism be for enabling/disabling the data?

d. Utilities

A first step at identifying common system utilities is done at this time. Standard data types are identified.

Products for Step 1. The following products will be generated during Step 1:

- a. Grouping of functions in the system level dataflow. This may be by drawing on the dataflow or by a list of top-level CSCs and the processes of the system level dataflow allocated to each.
- b. Requirements allocation matrix from SRS to top-level CSCs.
- c. Data item allocation matrix assigning data items from the SRS to top-level CSCs.
- d. Software Architecture Concepts paper describing the software architecture framework of the system.

Deliverables. SDD entries 1.0, 3.1 and 3.1.2 will be written. SDD 3.1.1 will be started. The Software Architecture Concepts paper and derived requirements will be kept in the CSC1 SDF section 2.12. The requirements allocation matrix will be part of SDD section 7. The data item allocation matrix will be part of SDD section 5.

Step 2 - Identify sub-level CSCs.

Sub-level CSCs primarily show the concurrency required in the system. Constraints due to hardware and the environment start to play a part here. A sub-level CSC may also be identified to package services. The data flow diagrams (DFDs) from the functional analysis phase are used in this process to determine functions and entities that should be sub-level CSCs. The process of transitioning the functional analysis into a design is called "recasting" the DFD. Some of the criteria that determine what constitutes a sub-level CSC include:

- i. Concurrency: Must the processing be done as a separately scheduled process? This is usually required to handle external events or because

processing must be done periodically.

2. Object oriented design: Is this entity an object which contains state data and provides operations on that data? Is this an entity which cannot be accessed simultaneously by concurrent processes?
3. Encapsulation and information hiding: Is this a service which will be used to make applications independent of the specific operating system or off-the-shelf software being used in the system? Note that these do not usually show up on a DFD but are required to insulate applications from dependencies on hardware or particular operating systems.

Guidelines on "recasting" the DFD. Steps to take to "recast" the DFD from the requirements analysis phase:

- a. Show handlers for external hardware interfaces. If an external interface is full-duplex (sending and receiving of data are independent), one may want to show a separate handler for each.
- b. Make data stores into objects.
- c. Raise processes from lower levels of the dataflow if they are necessary to understand major processing required. The product of steps a,b,c is called a preliminary entity diagram.
- d. Collapse processes into tasks by working inward from the external interfaces:
 - i. Can processes on the data flow be done sequentially because there is time to do it before the next input arrives?
 - ii. Should a data store become the state data of a process already on the data flow?

The product of this step is called the final entity diagram.

Products for Step 2. The following products are developed as part of Step 2:

- a. The final entity diagram for each top-level CSC showing the Ada tasks. The entity diagram may also show services/utilities, if they help tell the story.

- b. Description of environment assumed. This will include any operating system services or off-the-shelf software you are assuming in the design.
- c. Outline containing each top-level CSC and its sub-level CSCs. This will be kept in the Software Development Folder (SDF). For each sub-level CSC:
 - i. List of functions the sub-level CSC is to perform. This should be traced to the SRS, and to the derived requirements from the Software Architecture Concepts paper.
 - ii. Rationale for making the sub-level CSC a task or a service.
 - iii. SRS data items "allocated" to this sub-level CSC. These should relate directly to data flows on the DFDs and track where these data flows are in the design.
 - iv. Special processing: initialization, recovery, shutdown and error handling of this sub-level CSC. Error handling has two aspects. One is to implement the fault detection and recovery mechanisms described in the Software Architecture Concepts document. The other is to begin designing the Ada exception logic.
 - v. Interfaces: This will show which interfaces will be by Ada rendezvous (and hence show up as procedure specifications) and which will be by system message passing services or I/O services.

Deliverables. SDD 3.2.x will be written for each top-level CSC. The final entity diagram will be included in this section. SDD 3.1.1 will be completed.

The environment description and outline will be kept in the top-level CSC SDF section 2.11. As design progresses this paper may not be maintained since it serves as the precursor to information kept in deliverables.

Step 3 - Document purpose of each sub-level CSC.

The purpose of Step 3 is to formally describe each sub-level CSC using Ada as a Program Design Language (PDL). Much of the information will be found in Ada commentary. The PDL will be compilable and will establish the interfaces to each sub-level CSC.

Types of Sub-level CSCs. There are two types of sub-level CSCs. The first is an object. The documentation of an object sub-level CSC will include a description of that object in terms of an abstract definition of its state data, and a description of its operations on that abstract definition of the state data.

The second type of sub-level CSC is a process. It is best described as a classical (input, processing, output) entity. Usually the I/O is via system calls rather than procedure or task entry calls. Therefore, it cannot be described as operations on an object.

A sub-level CSC may be a combination of the two types. It may have some processing which is best described in terms of (input, processing, output) and in addition, encapsulate some state data upon which a user can operate via procedure calls.

Overview on objects. Following is a discussion about objects as viewed during preliminary design:

An object is a data type (or a collection of data) together with operations on that data. There are two views of the object: the conceptual view and the implementation view. At preliminary design we are concerned with the conceptual view. At detailed design we are concerned with the implementation view. The conceptual view is a description of the object as viewed by the user of the package. It must provide a clear and complete description of the object, and the operations on the object. Included in that description are any consistency rules which help describe the concept of the object in terms of data dependencies, uniqueness of elements, etc. The operations are specified as procedures and/or functions, together with a complete description of the effect of the operation on the conceptual view of the object.

Common conceptual views used are sets, queues, stacks, database tables. Often an object is not completely described by saying it is a set or a queue, etc. There are usually conditions that must be maintained to provide consistency. These are either interrelationships or limitations placed on the data. This information may affect the design of the user of object. We will call these conditions consistency rules. For example, an address book is a set of (name, address) pairs. A consistency rule for the address book object may be that for a given name, there is only one unique address. This limits the use a user may make of that address book object. For example, if a user wanted to put both a home address and a work address for each name, he would have to design some way of making the name unique (possibly by appending ".H", ".W", respectively).

Products for Step 3. The following products will be developed during Step 3:

- a. This information will be documented as Ada specifications and accompanying Ada

commentary, complemented by text in the SDD.

- i. For entities which are objects the conceptual view of the data will be described with Ada commentary in the package prolog. The operations on the data will be Ada specifications of procedures and/or functions. Parameters of the procedures will be documented in type definition packages, using the "TBD" type definitions package when refinement of the data type can be postponed until detailed design. The effect of the operations on the conceptual data will be described in the procedure/function prologs.
 - ii. For entities which are (input, processing, output) entities, the conceptual view of that processing will be described with Ada commentary in the package prolog.
 - iii. For entities which are a combination, both types of commentary will be present in the package prolog.
- b. The data dictionary will be updated with the mapping between SRS data items and their location in Ada packages. Note that they may map to Ada data types, Ada data packages, or even functions or parameters of procedures. Some may even not appear in the design if two processes are combined into one Ada procedure.
 - c. The requirements traceability matrix will be updated to show which sub-level CSCs implement each requirement in the SRS.

Deliverables. SDD 3.2.x.y will be written for each sub-level CSC. The preliminary PDL for the sub-level CSC will be included in this section. SDD 5. and 7. will be updated to show traceability to the sub-level CSC.

Step 4 - Describe abstract algorithm of each sub-level CSC.

This step supports development of the hardware resource allocation information, estimates of source lines of code and modeling of the system using a discrete event simulator.

Overview of abstract algorithm. Although we concentrate on the abstract view during preliminary

design, we must make sure the design will work. We therefore make simplistic assumptions about the implementation of each package so that a worst case scenario can be modeled. Modeling can help identify areas of the software that must be carefully implemented. Modeling can also verify decisions about allocation of the software to hardware, and will help develop the resource allocation information for the SDD. The abstract algorithm is oriented towards the processing required per input for each operation of the object, or for a typical input to a processing entity.

Products for Step 4. The following products are produced during Step 4:

- a. Describe implementation assumptions. It is assumed that system engineering has provided modelers with frequency of external inputs.
- b. Estimate number of Ada statements per step in algorithm for each operation of the package. This includes expected number of times through each loop. This estimate may be per reference if step is to be performed by calling on an operation of another sub-level CSC. Note that there may be a "background" task, not specified as an operation, but necessary to implement concept. For example: a garbage collection task which runs periodically to delete old entries in a database.
- c. Estimated source lines of code.
- d. Estimated memory requirements.

Deliverables. SDD 3.1.3 will be written. The algorithms and sizing estimates will be kept in the top-level CSC SDF.

Methodology for Detailed Design

Step 1 - Establish the "select" logic of each task.

The first step in detailed design is to outline the logic to be used in the "select" statement of each task. The concurrent entities identified in preliminary design are expanded upon. One sub-level CSC in preliminary design may turn into several tasks in detailed design. Tasks may be introduced for functions which are performed periodically, or to queue data to provide a looser coupling than the Ada rendezvous. Decisions as to which entities are callers and which are called are reviewed. This requires examining the entity diagram for tasks that communicate with the subject task, in addition to the abstract algorithm of the task. The relationship of execution order is also examined between tasks that interact in order to determine if there is a

potential for deadlock, or starvation.

Considerations. The following issues should be addressed during this step.

1. Is there a cycle in the usage of tasks (potential for deadlock)?
2. For selects with guarded entries: will at least one guard always be true (if not, PROGRAM_ERROR will be raised)?
3. Are task entry calls hidden by encapsulating procedures within the same package (to protect logic from user)? If not, will the use of timed entry calls, or an abort of the calling task, cause problems in this task's "select" logic?
4. Can the value of a guard change between the time it is evaluated and the corresponding rendezvous?
5. Will a steady stream at one entry block other entries of task?

Products of Step 1. The following products will be generated during step 1:

- a. Updated entity diagrams reflecting any changes in the caller/calling relationships.
- b. Ada package diagram of the internal relationships of the CSUs within the sub-level CSC.

Deliverables. SDD Section 4.x will be written for each sub-level CSC. The Ada package diagram will be included in this section.

Step 2 - Specify the implementation view of each S-CSC.

This step provides the information needed to eventually code each sub-level CSC. It also identifies the CSUs by step-wise refinement of the sub-level CSC.

Refining Descriptions of Processing Entities. For processing entities which are input, output, processing entities, specify the processing algorithm by step-wise refinement. Data type definitions for the input and output which were at an abstract level during preliminary design are now completely specified.

For processing entities that are objects:

1. Define the implementation of the state data of the object.

2. Define the implementation of the operations by describing them in terms of their manipulation of the actual object.

3. State the mapping from the implementation to the conceptual view. This mapping includes how initialization conditions of the object will be implemented and how the consistency rules will be maintained. There may be limitations placed on the implementation view which were not part of the abstract view. An example of this would be adding a maximum size to an unbounded set abstraction.

4. Verify that the mapping is consistent and complete:

- a. For every state of the implementation of the object, is there a corresponding state of the abstraction?
- b. Do implementation initialization and each implemented function maintain the consistency rules of the abstraction?
- c. Is the implementation initialization equivalent to initialization of the conceptual view?
- d. Is each implemented operation equivalent to the corresponding conceptual operation?

Products for Step 2. The following products will be produced during Step 2:

- a. PDL for package bodies, procedure bodies, and task bodies.
- b. PDL for the select logic in the task bodies.

Deliverables. SDD Sections 4.x.y will be written for each CSU. Section 5 will be completed.

Acknowledgement

The work described in this paper was performed by the Jet Propulsion Laboratory, California Institute of Technology, under the sponsorship of the U. S. Department of Transportation, Federal Aviation Administration, through an interagency agreement with the National Aeronautics and Space Administration.

References

1. United States Department of Defense, *Military Standard Defense System Software Development*, DOD-STD-2167A, 29 February 1988.
2. United States Department of Defense, *Data Item Description - Software Design Document*, DI-MCCR-80012A, 29 February 1988.
3. Booch, G., *Software Engineering with Ada*, 2d ed., Benjamin/Cummings Publishing Company, Inc., 1986.
4. Cherry, G. W., *The PAMELA Designer's Handbook - Vol. I & II, Thought** Tools*, 1986.
5. Nielsen, K., and Shumate, K., "Designing Large Real-Time Systems with Ada", *Communications of the ACM*, Vol. 30, No. 8, August 1987.
6. Gomaa, H., "A Software Design Method for Real-Time Systems", *Communications of the ACM*, Vol. 27, No. 9, September 1984.
7. Ward, P. and Mellor, S., *Structured Development for Real-Time Systems*, Yourdan Press, 1985.
8. Seidewitz, E. and Stark, M., "Towards a General Object-Oriented Software Development Methodology", *Ada Letters*, Vol. VII, No. 4, Association for Computing Machinery, Inc., July, August 1987.
9. Firesmith, D., *Ada Project Management*, 1987.
10. United States Department of Defense, *Data Item Description - Software Requirements Document*, DI-MCCR-80025A, 29 February 1988.



Dr. Karen S. Ellison has over 18 years experience in software development and maintenance in both commercial and aerospace applications. She is currently on contract at JPL to provide day-to-day guidance to the FAA Real-Time Weather Processor project on the practical use of software engineering methodologies for real-time, distributed systems implemented in Ada.



Mr. Goulet has over 20 years experience in the development of information systems and embedded realtime applications. He is currently the Software Development Manager for the Realtime Weather Processor Project at the Jet Propulsion Laboratory in Pasadena, California.

The mailing address for both Dr. Ellison and Mr. Goulet is: Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Mail Station 506-447, Pasadena, CA 91109.

**LESSONS LEARNED IN THE PREPARATION OF A
SOFTWARE ENGINEERING EXERCISE
(A Life Raft at SEE)**

John P. Fitzgibbon and Catherine Peavy

Martin Marietta Information and Communications Systems*

Abstract: Potential DoD contractors are now required to demonstrate their ability to successfully develop software prior to contract award. One mechanism used to assess a contractor's capability is to complete a Software Engineering Exercise (SEE) defined by the procurement agency. This paper relates the experiences of Martin Marietta Information and Communications Systems during a Software Engineering Exercise that was part of a competitive contract procurement. This paper summarizes the experience as lessons learned.

INTRODUCTION

Software systems are expensive, complex and often fail to meet the basic needs that instigate their development. Procurement agencies, cognizant of these problems, have been concentrating on ways and means to mitigate these risks through more careful evaluation of prospective contractors. Several recent developments indicate that these agencies are switching emphasis from the product to the process that creates it. Perhaps the most significant development is a technical report¹ by the Software Engineering Institute describing a method for assessing a contractor's ability to produce software. Commissioned by the Electronics System Division (ESD) of the Air Force, this document typifies the increasing interest by the government in the process of system development.

One of the most significant aspects of DoD-STD-2167² is the requirement that automated requirements analysis tools be used to develop models of the software requirements. These models promote understanding and lend themselves to analysis. Another trend affecting the software engineering process is the mandate³ to use Ada for design and implementation of new systems. This

leads to emphasis on modern software engineering principles and supporting development environments that perform evaluation and verification of software engineering products. The Software Engineering Institute (SEI) has recommended that contractors be required to demonstrate their capability to undertake software development prior to the commencement of the contract. One of the methods recommended for evaluation is a Software Engineering Exercise (SEE).

The declared purpose of the SEE is to provide the procurement agency with an opportunity to evaluate the offeror's software development methodology. The offeror is required to analyze a software system, propose a viable design and then to defend the solution before a panel of application experts assembled from industry, government service and academia. This allows the agency to make an objective appraisal of the quality of the product, and the efficacy of the development procedures and practices. Furthermore, it allows the procurement agency insight into management effectivity.

The problem for this exercise involved developing a Real-Time executive that would support the control and execution of a distributed simulation. The offeror was directed to provide a complete software architecture outlining each major design component, to perform detailed software requirements analysis on at least two of the major architectural components, develop a top-level design for the analyzed components and to provide a detailed design for at least one element. The executive was to be designed in such a manner that it would be implementable in Ada. Furthermore, the design of low level software elements was to be expressed as compilable Ada PDL. The instructions to the offeror also stated that documentation should be representative of the offeror's software development methodology. Although the

documentation could be informal, the instructions implied that information provided by a Software Requirements Specification (SRS), a Software Top-Level Design Document (STLDD) and a Software Detailed Design Document (SDDD) was expected.

Following submission of the offeror's engineering documentation the procurement agency evaluated the solution. Once the agency had evaluated the submitted data, a review was scheduled to provide the offeror the opportunity to describe the results of the engineering analysis and design. The offeror was allowed two hours to present analyses, interpretations of requirements, design descriptions and to describe alternative design approaches. The presentation was followed by a five hour examination of the requirements analysis, software design and the participants in the exercise. The remainder of this paper recounts the experiences that Martin Marietta had during the performance of a Software Engineering Exercise.

LESSONS

While our efforts on the SEE were successful in the acquisition of the contract, we feel that with better preparation our results could have been better. The following sections detail what we feel are the most important lessons derived from the execution of the exercise. Although the lessons were particular to the SEE, some reflection shows that they are applicable to the Software Engineering process in general. These lessons represent management principles but they are rendered from the perspective of a software engineering practitioner. Software Engineering managers should carefully consider these lessons so that they may better understand the implications of their actions on their staff, the product and the process of development.

Never try to pick up two watermelons at the same time

The prospective bidder is faced with two different problems in the preparation of a Software Engineering Exercise (SEE). The first problem is the analysis of the problem and synthesis of a design. However, the second and more important problem is to select and apply a suitable development methodology to the development process. Most companies have established software engineering methodologies that have evolved over a number years in response to contract experience. It is a very rare case when this experience base includes a dozen or more large scale Ada

development projects. Therefore, most companies are in the unfortunate situation of having to update their current practices and procedures to accommodate Ada and DoD-STD-2167 requirements. Since the scope of this task clearly exceeds the time limitations imposed by a competitive procurement, it is necessary to work out the software methodology in advance.

Now that the contractor has defined a modern software development methodology it *only* remains to assemble a team. This team must be familiar with the application area and skilled in the application of the selected software development methodology. There may be companies rich with an abundance of highly skilled software professionals, couched in the most modern development methodologies, who are instantly available to respond to proposals. (If there are such, they would be immediately disqualified out-of-hand if the procurement agency suspects that "ringers" have been brought in.) For the rest of us, we have no alternative other than training our staff in the principles, techniques, and tools.

Participants in a Software Engineering Exercise (SEE) should provide a Software Development Plan (SDP) as one of the deliverable documents even if it is not specifically required. Submission of a SDP provides the contractor an opportunity to explain the software methodology used during the SEE. The real purpose of the SEE is to showcase the software development methodology rather than the product. The proposal section of the SEE response should relate how application of the selected development approach lead to the resultant solution to the software engineering problem.

We recommend that the SDP be prepared in advance due to the time pressures inherent in a competitive procurement. This does not imply that the contractor has a single methodology that it applies to each of the projects undertaken by the offeror. On the contrary, we suggest that the offeror develop a standard set of procedures, encompassing several development paradigms, that may be selected and integrated into a software development methodology that is tailored to the specific needs of a particular project. The SDP is in effect, a template referencing a set of possible techniques and procedures that may be selected to provide several variant sequences of development activities each of which is appropriate for a given development paradigm. Along with the set of procedures is a set of rules that provide constraints on the various combinations of procedures. For

example, there may not be a proven methodology that combines Object Oriented Analysis, Jackson System Design method and Top-Down system development.

The real lesson is that without establishing software development procedures, techniques and methods in advance, the application experts will not be effective in solving the designated engineering problem. Next, the development team must be trained in the methods and procedures composing the methodology. Finally, this system of engineering processes must be proven in actual practice since the methodology will be evaluated by the quality of the resultant product. Understanding that the system problem space is coupled to the engineering process space is critical in assuring success of the development effort.

Do not try to eat the elephant in one bite

Problem descriptions used during a SEE tend to have very large scope and be somewhat ambiguous. It is not unusual for there to be conflicting statements within the problem definition. Furthermore, the offeror can be prohibited from asking questions regarding the meaning of the problem statement or other requirements specifications. Given these conditions it is difficult to control the engineering process. We believe that to some extent this is a premeditated action on the part of the procurement agency designed to evaluate the offeror's management ability. Therefore, the offeror should interpret the requirements so that a solution may be achieved within given time and available resources.

The SEE instructions provide the offeror considerable latitude in requirements interpretation and the scope of problem analysis. The key to success is to develop a consistent strategy for the course of analysis. The first premise is that not all aspects of a problem are equally important. Therefore, the analysts must prioritize different problem aspects, high-lighting the most critical ones. The analysis in these areas should progress first and to the greatest depth. For less important areas the analysis should be deferred until the rest of the problem is better understood. For those less critical areas, the analysts should make assumptions that will simplify the resultant design. This strategy leads to systems that are simpler, more understandable and ultimately easier to operate and maintain. In the context of the SEE, this leads to a problem definition that is tractable within the available time and resources.

Some readers may argue that the offeror does not have the right to make interpretations of the system specification or to decide what is more or less important. However, ambiguous specifications are common in our industry and are often the result of ambivalence on the part of the customer or an attempt to reach consensus between different factions. By providing a "strawman", the dialogue between the customer and the contractor is transported from an abstract plane to specifics. The contractor can present alternatives that may not have been feasible when the need for the system was first realized and specified. Moreover, this allows the analysis process to proceed to closure.

Each journey begins with the first step

The purpose of the SEE is to demonstrate the effectiveness of the offeror's software development process. An important part of this process is the generation of engineering data products. Although the instructions to the SEE problem allow the offeror to provide informal documentation, we chose to develop documentation as required by DoD-STD-2167. By doing so we provided documentation that would be representative of that prepared under any future contract with this customer. Furthermore, this reinforces our premise that documentation is a normal product of the software engineering process rather than an obligatory and onerous task that has nothing to do with the process. However, the first step in the engineering process is to analyze the total system (i.e., software hardware and manual operations) and define how it will be used.

We elected to develop a System Segment Specification (SSS) and an Operational Concept Document (OCD) in addition to the software specifications and design documents. We believe that the Systems Engineering analysis and the Software Engineering Analysis are interdependent processes that form a unified development methodology. The Systems Engineering process generates an essential understanding of the context surrounding the software product. Without performing adequate Systems Engineering analysis prior to software engineering, designers have little guidance in the development of a realistic solution.

One of the most important products of the System Engineering Analysis is the Operational Concept Document and as such requires special attention in this discourse. Consider a distributed system architecture comprising a central host that processes information and a network of satellite workstations

that display the information and provide system control. Should the system design employ centralized control or distributed control? Should the workstations be uniquely configured to perform a specific subset of functions or should each workstation provide all of the functions that are required by the set of all possible users? Should the human machine interface be designed to accommodate the least trained user or should responsive performance be emphasized or must the interface exhibit either behavior depending upon the requirements of the user? Is it necessary for the system to operate in more than one mode concurrently? Should different system modes be physically partitioned so that there is no chance of confusion or cross-over? The Operational Concept Document provides a framework to evaluate these design alternatives by defining who will use the system, how it will be employed and deployed and which system requirements are satisfied by manual procedures.

Bigger is not better

One of the most frequently recurring patterns of organization among successful projects is a small design team that participates in all phases of development. Frederick Brooks⁴ described the chaos that resulted from trying to employ a huge team of 1,000 software programmers to design OS/360. Little progress was made until the design became the responsibility of a 12 member architecture team. On the contrary, Harlan Mills demonstrated that a small team of specialists could design a system of similar size (New York Times operating system⁵) in a shorter time interval and with higher product quality.

The team we selected consisted of a small kernel of individuals who were well trained and had been working together for over a year prior to the SEE. We recommend that the team comprises specialists in each of the skill categories related to the project. Skill categories might include a language specialist, a data base designer, an application expert, a requirements analyst, and a system designer.

Another category that is becoming increasingly important is a software engineering tool specialist. This individual is an expert in the engineering techniques, analysis methods and the tools that support the constituent parts of the selected engineering methodology. In a long term project, this individual may also play the role of a software engineering toolsmith tasked with automating portions of the process. If we had had such a

specialist during the SEE, we would have better exploited the capabilities of our engineering tools and had better productivity.

Modern workstations, analysis tools, word processors, graphics tools, and other facets of desk top publishing are of limited use if the staff and management are unaware of their capabilities and unable to exploit such tools fully. With this in mind we recommend that the development staff be supported by a complement of administrative aides. This support team should comprise commercial artists trained in computer graphics, word processing clerks capable of editing technical manuscripts, and engineering aids trained to execute engineering procedures effectively. The supporting staff should work with the development team rather than be quartered in a clerical pool. During the SEE, our support staff was located at some distance from the development team. Consequently, we had to use engineering talent for several clerical tasks. This effort would have been better employed doing engineering tasks.

Once the team has been selected, management must trust them to accomplish their task through self-direction. Management needs to monitor progress and assure that resources are available when they are needed. In most cases, the team members are the ones most qualified to decide what resources are needed and how they should be applied.

Teach old dogs new tricks

Although the team should be self-directed to the greatest extent possible, management must retain visibility into the development process. A development team can easily lose track of its primary objectives by becoming too engrossed with tangential issues and minor details of the problem. At this point management must be capable of stepping in and moving the development back on track. To do this Engineering Management must understand the methods being used by the design team and develop methods and metrics to monitor the on-going process. This means that training for Engineering Management is a critical priority and that this training must include modern development methodology as well as management issues.

Engineers find it difficult to separate the process of requirements analysis and design. We believe that while a partial cause for this phenomena lies with inadequate training and personal disposition, another cause is the fact that the two processes are interdependent. Rather than inhibit the creativity of

the developers by enforcing an arbitrary separation of the processes we decided to allow iteration between the two processes. The team was allowed to iterate freely between analysis and design with the stipulation that their perambulations were recorded in engineering notebooks. These notebooks provided the source material for the engineering data products required by DoD-STD-2167.

The conversion of source material to standard documentation format was done by a small team of specialists acting as technical writers. This allowed us to uncouple the design process from the recording process. It is tempting to suggest that the writing team be composed of non-engineering talent so that labor costs can be reduced. However, we would prefer to use experienced design engineers. We discovered that the design team was not always rigorous in the maintenance of the design notebooks and that the writing team had to interact with the designers to get necessary information. The resulting dialogue forced the design team to consider the validity of the candidate design and to consider possible alternatives. If the writing team had not been composed of top quality software engineers, the dialogue might have degenerated into a monologue and have added little value.

Substance before style

The most important thing a participant in a Software Engineering Exercise can do to improve the quality of their response is to check the submitted documentation for consistency. During the SEE we learned to appreciate the value of using Engineering Analysis tools to verify interfaces between software components. We prepared requirements traceability matrices mapping system requirements to design components. Due to the limited scope of the problem we were able to perform this task manually. However, we learned that this becomes unmanageable for any problem that is realistically sized. Since that time we have directed some research effort into developing generalized data base models to generate the necessary traceability reports automatically.

During the SEE we had different groups working on the requirements analysis and system design. This enabled parallel development to some extent while on the other hand it created problems with consistency of documents. This is essentially no different than development under traditional life-cycle paradigms where requirements analysis and design are partitioned by time if not personnel. We were able to control this parallel development by

the requiring that the two groups exchange documents for review. This interchange of information improved the work of both groups and promoted consistency of documentation style, form and content.

While attention to style and format can make a good product sparkle, it must never be substituted for accuracy of content. In an effort to enhance the polish of our submitted documentation, we passed up a final opportunity to review the documents for technical consistency. One of the most painful experiences of the SEE for our team was the discovery of errors after submission of our response. These errors were the direct result of polishing an incomplete product.

Measure your bucket before bailing

Performance analysis should be started as soon as a candidate software architecture is sketched out. When there are no Quantitative Performance Requirements (QPR's) provided with the problem statement, the engineering team should make assumptions regarding reasonable performance based upon experience and similar systems. The Operational Concept will often provide valuable insight into what the customer expects of the system. Using mathematical analysis or simulation it is possible to quickly determine performance for the major system behaviors.

There are certain figures of merit that can be derived by simple means that provide valuable information on expected system performance. The first figure is the mean time for task context switch. This can be computed with an Ada program consisting of two or more tasks. The program is written so that the tasks rendezvous with each other in a circular configuration (e.g., $A \Rightarrow B \Rightarrow C \Rightarrow A \dots$). A loop of 1,000,000 is appropriate for machines that execute at the rate of one million instructions per second (1 MIP). Task rendezvous rates of 1000 per second are typical for such an environment. A variant of this figure is the mean time for process context switch. This is determined in a similar manner by having processes communicate rather than Ada tasks. This is especially relevant for systems that employ multiple-processor architectures.

Another necessary figure of merit is the maximum effective data rate for each data link in the distributed system. This can be determined by a simple program consisting of two cooperating processes that continuously exchange messages

without performing any information processing. The effective data rate should be plotted as a function of average message size for each data link in the system. For data links that use multiple access channel acquisition (e.g., Ethernet) it is important to parameterize the effect of channel contention on performance. This can be done by testing pairs of cooperating processes or developing a test program variant that communicates with an arbitrary number of clients.

The results of a performance analysis may be disturbing to the design team. As a result of this analysis, we determined that no more than two task rendezvous could be tolerated for a single transaction. We modified our design accordingly and used buffering to minimize communication overhead. Regardless of the outcome, we feel that performance analysis data should always be presented as part of the response to the SEE problem. A perceptive customer will expect it to be there and question its absence.

Practice makes perfect

If there is one thing that we wished that we had done in preparation for our first Software Engineering Exercise it would have been to practice in advance. Ideally, we would have chosen a design team and practiced by performing mock SEE's internally. This would promote self-confidence on the part of the design team as well as pointing out deficiencies in time to take effective action. Even if the team has been thoroughly trained in the principles of software methodology, there is no substitute for actual experience with a SEE.

Having gone through a SEE we have gained considerable respect for its ability to point out problems with the software development process and evaluating the effectiveness of a design team. We think that this technique should be used to evaluate new software methodologies, research and development projects, to validate new software standards and practices, and to evaluate software engineering tool products. In addition to self-evaluation, we think that the SEE format is particularly effective in predicting the performance of sub-contractors. Furthermore, experience gained applying this technique to others will provide valuable knowledge and insight on the most effective way to execute a SEE as part of a competitive procurement.

Money for nothing and checks for free

The contractor should consider the statement of the SEE problem very carefully before beginning to frame a response. The procurement agency will surely have chosen what it considers to be the most difficult technical problem for the *real* system that is being procured. The government is understandably trying to get a head start on the development of his system during the competitive phase of the procurement. In doing so, he is dropping clues regarding what he considers to be an appropriate answer. The nature of the SEE problem statement exposes the government's *real* for technical evaluation. The mainline proposal team can and should use this intelligence in framing their proposal. Similarly, the SEE team should examine the RFP and the proposal for guidance in their response. The SEE team should choose a design approach that can arguably be extended to work with or be part of the "Real" system. Their response should emphasize commonality shared by the "Real" system and the SEE system.

An ounce of prevention has more value than a ton of remorse

Once the SEE team has submitted their final draft, it is time to start preparing for the Review panel. The first step in this preparation is to have each member of the SEE team read every document that was submitted with the response. This should be a critical review where each team member is trying to discover flaws in the approach or documentation.

While reviewing the submitted documents, the SEE team should make a list of deficiencies, missing data, errors and other questions that are not adequately addressed. The SEE team should then try to provide written answers to each question on the list. If an error was simply an oversight then it is best to admit it and to say that it was identified during the post-submission review. Participants of a Software Engineering Exercise are very aware of the unusual time pressures resulting from this situation. In our own experience we successfully predicted approximately half of the technical questions presented at the review.

Another task is to find out as much as possible about the team of "Grey Beards" who will be on the review panel. Several of them will be academics; reading their recently published papers will provide insight into what they will ask. The other members of the review panel will be military and civilian

employees of the procurement agency. In most cases, someone in your organization has dealt with them before and can define the hot buttons. The best way to prepare the SEE team for the actual conduct of the "Grey Beard" review is to stage a dry run. The contractor should select a Technical Review team consisting of senior technical staff and executive level managers from other projects. The Technical Review team will play the role of the "Grey Beards" during the dry run. To aid the Technical Review team in preparing, the SEE team should provide the list of questions and answers generated during the document review. These questions will provide the reviewers with an introduction to the SEE problem and the critical aspects of its proposed solution. Moreover, it will stimulate the thought processes of the review panel in defining their own questions. If the SEE team can survive such an inquisition then it should be well prepared for the actual review.

After the dry run, the SEE team and the Technical Review team should meet for a debriefing. The purpose of the debriefing is to assess the preparedness of the SEE team. The Technical Review team should provide answers to any questions that the SEE team failed to answer adequately. The SEE team should also ask the Technical Review team to help answer any unresolved questions. The management members of the Technical Review team tend to be very adept at providing answers to impossible technical questions.

The "Tools 'R Us" Syndrome

Too many discussions of software engineering methodology begin and end with tools. Available tools far too often define and mandate the choice of a particular methodology rather than the intrinsic nature of the problem under study. Lack of adequate tools to support a particular methodology may discourage engineering management from applying the most appropriate methodology. On the other hand, the desire to use the latest hot methodology can lead to a precipitous choice of engineering tools based solely on the felicitous assertions of a CASE vendor.

A software engineering methodology comprises a development strategy, techniques for analysis, tools, standards, metrics and appropriate training. Success requires that each item be addressed carefully and that none be shorted. In particular, the training budget should never be sacrificed for the benefit of any other aspect of development. Time and again,

dollars spent on training have had the greatest pay-back. Dollars saved on training have borne the greatest cost.

A development strategy or paradigm should be the most carefully considered decision of any engineering process. The development strategy is the pattern that unifies the analysis methods, tools, standards and metrics that are applied to problem resolution. If an Object Oriented methodology were employed, we might expect the choice of such engineering disciplines as Object Oriented Analysis, Information Modeling, and Object Oriented Programming practices. To support these methods we would require an Object Oriented Language and tools to develop and analyze Entity Relationship Diagrams. Standards and metrics, supported by other tools, would instrument the engineering process and aid in the evaluation of the software products. No matter what paradigm is chosen, it is critical that it be complete. Each method, tool and procedure must be compatible with and complement each element of the methodology.

Techniques or methods are forms of analysis that describe some aspect of the problem under study. Usually these techniques employ engineering models that emphasize the attributes under study while diminishing or ignoring other aspects. Since these models and techniques focus on singular attributes it is not possible for any one method to be comprehensive. A given methodology may incorporate Structured Analysis, Structured Design, Structured Testing and Top-Down development. This is not the only feasible methodology nor is it appropriate for all projects. These methods need to be selected based on the nature of the problem under study and the personnel performing the analyses.

Tools support a specific technique by providing a mechanism to depict and analyze the engineering model. Structured Analysis uses a model comprising a set of Data Flow Diagrams (DFD's), a data dictionary and a set of process specifications. The tool must support each of these aspects of the model and aid in its analysis. At a minimum, it must be able to cross-validate the DFD's, the dictionary and the Input/Output declarations of the processing specification. Furthermore, it must be able to verify consistency of expression between different levels and partitions of the set of DFD's. Other useful capabilities include normalization of data flow definitions within the data dictionary, reports that specify data elements crossing interfaces, mechanisms that allow capture and

retrieval of auxiliary data attributes (e.g., range, accuracy ...) and analyses of change activity on different portions of the model. The point of the foregoing discussion is to demonstrate that the outputs of tools should be verifiable and that they should directly map to products required by the engineering process.

Standards provide a set of criteria by which to compare the quality, consistency and completeness of engineering products. Most projects have coding standards that prescribe presentation format, usage of language features, and naming conventions. However, far fewer projects have standards that address the partitioning of requirements models, allocation of requirements to components, appropriate levels of design decomposition, detail requirements for process specifications or provide criteria by which to evaluate test cases. For a methodology to be effective, standards must exist to evaluate each product and activity included in the methodology.

Metrics provide a standard to measure the state of the engineering process. Most metrics that have been defined relate only to the implementation activity. The most often cited metric is the Line of Code (LOC) statistic. Most managers can quote program productivity rates (e.g., 2 LOC/hour) but few have any idea of what acceptable error density rates might be (50 per 1000 LOC?). Furthermore, how can we measure other engineering activities such as requirements analysis or test? Good metrics should provide management with answers to the following questions. How far have we gone? How much further must we go? At what rate are we approaching completion? Suitable metrics for each engineering activity and process must be developed and applied in the context of the tools and methods that compose the selected engineering methodology.

CONCLUSIONS

This paper recounts the experiences of Martin Marietta Information and Communication Systems while performing a competitive Software Engineering Exercise. However, many of the lessons learned are applicable to the management of any engineering process. To succeed in coming years companies will require a large number of well trained, dedicated engineering professionals. The number required and the market demand for such individuals implies that it will not be feasible to hire all of them. Instead, companies must develop training programs for their existing work force.

The software engineering process is changing rapidly due to growth of requirements and revolutionary advances in technology. To cope with and to understand the nature of this change, companies will have to invest in the acquisition and transfer of modern engineering technology. This requires research into software engineering processes, the interdependencies among analysis methods, standards to evaluate engineering products and metrics that instrument the process of development. The results of this research must be codified into new engineering practices, standards and procedures. Once developed these engineering practices must be validated through real world application by projects. Feedback from the projects will assure that research programs are investigating the most important problems and that new procedures are effective.

The engineering process can no longer be viewed as separate activities performed by isolated specialists but rather as a single process executed by a team of generalists. The members of future development teams will be schooled in all the skills necessary to engineer software systems including analysis, design, implementation, test and integration. The same team will be responsible for the product from concept evaluation through deployment. Consequently, staff loading will be more stable throughout the project life and productivity will be higher. The impact to the product will be higher quality, lower cost and sooner availability.

The development processes must become more flexible than current paradigms allow. Rather than a single methodology, based on a standard paradigm comprising a small set of methods, modern methodologies will comprise a rich set of various disciplines. These methods will form sets of interchangeable process components. The components will form an integrated system bound by a paradigm that is tailored to specific product features and program needs. In this manner, different methodologies can be applied to accommodate technical risk, project resources and applicable technology.

To succeed, companies and employees must exhibit an unwavering commitment to professional excellence. Employees must be trained and nurtured to bring out their full potential. Employees must be willing to dedicate considerable energy to learn, master and apply new technology as it becomes available. Methods for monitoring employee development and rewarding progress must be instantiated. Furthermore, companies must provide an environment in which employees can excel and they must be unwilling to accept anything other than superior performance.

¹"A Method for Assessing the Software Engineering Capability of Contractors", Software Engineering Institute, CMU/SEI-87-TR-23, ESD-TR-87-186, Preliminary Version, September, 1987.

² Defense System Software Development, DoD-STD-2167, 4 June 1985, section 5.1.1.7, p. 27.

³"Computer Programming Language Policy", Department of Defense Directive, Number 3405.1, April 2, 1987.

⁴Brooks, Frederick P., *The Mythical Man Month*, Reading, Mass.: Addison-Wesley, 1975.

⁵Baker, F. T., "Chief Programmer Team", *IBM Systems Journal*, Vol. 11, No. 1, 1972.

John P. Fitzgibbon

Mr. Fitzgibbon is a senior staff engineer for Martin Marietta Information and Communications Systems. His current assignment is in Colorado Springs supporting the United States Space Command in its Granite Sentry program. In this capacity he is performing system design analysis and defining engineering processes to support the development of software. Mr. Fitzgibbon is a member of the Institute of Electrical and Electronic Engineers. He received a B.S. in Electrical Engineering from Northwestern University in 1972. He has studied Computer Science and Computer Engineering in graduate level programs from the University of Iowa, the University of Minnesota, the University of South Florida and the University of Colorado. Mr. Fitzgibbon can be reached at 4180 East Bijou, Colorado Springs, Colorado 80909 or telephoned at (719) 591-3800.

Catherine H. Peavy

Ms. Peavy is the manager of Advanced Software Technologies for Martin Marietta Information and Communications Systems Company. Her responsibilities include the management of the development of Software methodologies and tools for the company, as well as managing the staffing, technical and career development for 150 engineers. Additionally, Ms. Peavy is a member of the corporate Technical review team for project and proposal efforts. Ms. Peavy is on the Board for the Annual National Conference on Ada Technology and a member of the ACM SigAda. She received a B.A. from the University of Colorado, Boulder, Colorado. She has done post-graduate work at the University of Colorado in Business. Ms. Peavy can be reached at P.O. Box 1260, Mail Station 0720, Denver Colorado 80201 or telephoned (303) 977-2370.

*The opinions expressed in this paper are those of the authors, and not necessarily the opinions of Martin Marietta Corporation.

A Comparison of Methods which Address the Development of Real-Time Embedded Systems.

R. Guilfoyle[‡], R. Pirchner[‡], L. Van Gerichten[‡], M. Ginsberg[‡], D. Clarson[‡]

[‡]Monmouth College
West Long Branch, N.J. 07764

[‡]Teledyne Brown Engineering
151 Industrial Way East
Eatontown N.J. 07724

ABSTRACT

A series of descriptions of software development methods, based on data obtained from their suppliers, is presented as an example of a basis for comparing methods. In this approach, methods rather than their products are compared. The methods described were selected because they are intended to help software developers who are faced with those concerns typically found in real-time embedded systems. Descriptions are provided based on responses to selected questions from a developer's survey. The authors followed the thesis that each end user must rank methods according to requirements that are unique to that user, and avoided making judgments on behalf of the user. Several trends observed by the authors are listed at the end of the article.

1. INTRODUCTION

The adoption and use of methods represents one of the major factors affecting software development during the past 15 years. During this period, a large variety of methods has been introduced, and it is natural that attempts should be made to compare these methods, and to offer criteria by which a software development organization can judge which methods are best suited to meet the needs of the organization.

This article demonstrates a technique of presenting information about methods which forms a basis by which comparisons can be made. It is not the intention of the authors to formulate an evaluation of the methods presented; rather, the article serves as an example of how the software engineering community can be provided with information for judging the suitability of a method to the individual needs of the development organization.

The authors have adopted the thesis that "absolute" rankings of software development methods are neither possible nor desirable. Moreover, it is the authors' position that the needs of the software development community are best addressed by providing

information that can help members of the community make their own determination of how closely a given method fits their needs. This fitness factor depends upon the problem domain for which software is being developed, upon the background and needs of the development team, and upon the environment in which the software is to be developed. This environment encompasses the management style and maturity level of the organization, as well as the physical facilities of the development organization.

Thus, in order to judge the suitability of a method for the needs of an organization, information for making such judgments must be readily available. As an initial step in providing such information, the authors have worked during the past two years on the development of a catalog of software development methods. In order to provide a ready basis of comparison, the catalog employs a uniform style of presentation across all methods, as well as a tabular format to contrast methods. In this article, the authors have provided some examples of the type of information presented in this catalog.

2. BACKGROUND

The first edition of the Software Methodology Catalog [Maha87] was developed for, and published by, the US Army CECOM installation located at Ft. Monmouth, New Jersey. Information for this catalog was gathered primarily by surveying method developers, though some additional information was solicited from users of methods in the software engineering community. The catalog contains descriptions of 47 methods and is available through the Defense Technical Information Center (DTIC).

During 1988, a continuation of the project involved the production of a second edition of the catalog based on a revised version of the developer survey. This edition, when completed, is expected to contain information about more than 60 methods. The project also included a second task in which guidelines were

proposed for approaches to the task of evaluating software development methods.

In the initial phases of the 1988 project, an analysis was made of the results of the first survey effort. Based on this analysis, an extensive revision was made of the questionnaire, with emphasis placed on soliciting more specific information on various method aspects. Additionally, an extensive analysis was made of the attempt to gather data from method users. The authors concluded that gathering meaningful data about individual methods from users requires information about the respondents themselves, and would involve significantly more resources than were available in the project. Accordingly, it was decided to use only developer/vendor information as a basis for the second edition of the catalog. The revised developer survey was distributed during the months of September through November, 1988.

Using information received from this survey for six of the methods, this paper presents a sampling of the basis for comparison which is employed in the catalog. The six methods selected are suited for the development of real-time, embedded systems. Such systems represent a principal application area for the Ada community, and the methods are representative of those currently being used for Ada development. Included are mature methods, as well as methods which have evolved in the last few years. In addition to the appropriateness for real-time development, the other criteria for selecting the methods for this article was the availability of survey responses at the time the article was prepared. The reader is cautioned that the inclusion of these methods in this article does not constitute a recommendation. Furthermore, before making evaluative judgments, it is evident that similar information about other methods should be reviewed.

3. OVERVIEW OF THE SIX METHODS

In this section, a brief overview is given for each of the methods in order to provide the reader with a general frame of reference.

3.1 ADM -- Ada Development Method

The developer of ADM, Donald Firesmith, characterizes this method as recursive; i.e., as cycling through small parts of the design, code and test activities. An early version was used in the development of the Advanced Field Artillery Tactical Data Systems (AFATDS) in 1985. The method has both data-flow and object-oriented foundations and deals with most activities associated with development.

Important steps to be followed include the identification of *Assemblies* and *Builds*; moreover, there are steps dealing with the staffing and scheduling of assembly development. Subassemblies of each assembly are recursively elaborated in the manner outlined above. Part of this elaboration process includes producing documentation as well as invoking configuration management. Some of the steps for dealing with subassemblies include storing their requirements in a method-specific library, developing method-specific diagrams and creating logical designs. Subassembly testing and integration are planned activities addressed in the method.

3.2 DCDS -- Distributed Computing Design System

The DCDS method is a collection of procedures and tools intended to deal with several phases of the software process, including the representation of the system, representing software requirements, design and testing. Its principal developers are Mack Alford and Loyd Baker; it was first used at TRW in 1973.

The initial step in the method is definition of system-level requirements using a method-related language for that purpose. Definable entities include functions, their inputs and outputs and their decompositions and allocations to hardware components. Beyond those, interface designs, control functions dealing with resource management, and fault tolerance are definable with the language. A method-related software requirements engineering technique is employed to decompose the high-level definitions to a state-machine stimulus-response level.

Several additional method-related tools and techniques are used to ensure consistency, establish interfaces, develop modules and package the implementation. In particular, there are steps for dealing with distributed processing, concurrency, scheduling and memory constraints. Finally, there also are tools and techniques for developing integration tests. Further information is available in the bibliographic references [Alfo87] and [DCDS87].

3.3 MASCOT -- Modular Approach to Software Construction, Operation and Test

The MASCOT method deals with concurrency, distributive processing and real-time issues from the onset of the software process. The method also identifies interfaces and data objects as part of its approach towards implementation. It started out in the United Kingdom in 1971 at Malven and has been upgraded several times; Ken Jackson and Hugo Simpson are its principal architects.

The starting point of the method is a "Design Proposal" that identifies the major functional subsystems and top-level internal data stores. Top-level units are decomposed into lower level ones by identifying active, passive and device dependent components. The developing network of units has its components defined in terms of data-flow, functionality and access interfaces. Network decomposition ends when simple elements, i.e. those that can be directly implemented in a programming language, are identified. In the process of elaborating designs, MASCOT employs several templates that have been devised to match a variety of circumstances. The method includes rules to be followed to guarantee that the simple elements conform to the architectural structure. There also are aspects of the method that deal with testing. Four articles covering MASCOT were published in a special issue of the *Software Engineering Journal*, May, 1986, jointly published in London by IEE and BCS. Additional information is available from the Defense Research Information Center, Glasgow, Scotland.

3.4 OOA -- Object Oriented Analysis

In 1987, Boeing Computer services initiated a project resulting in this method; Mark Smith and Stephan Tokey are its principal developers. The method was first used for the development of a deliverable system in 1988. The method calls for the specification of the requirements of a system in terms of essential object classes. The product of the method is a statement of requirements that is intended to be then input to an object-oriented design.

Three essential activities make up OOA:

1. Identification and Specification of Object Classes
2. Specification of Object Communication
3. Identification of Class Operations.

The method is an extension of existing methods which employ Information modeling, Data-Flow modeling, and Finite-State modeling. There is a greater emphasis in the method on object/class communication and interaction which is represented in the Object Class Operation model. A more complete description of the method is available in [Smit88].

3.5 SEM -- System Engineering Methodology

This method is a collection of techniques for dealing with system requirements, developing specifications, developing prototypes and developing architectural designs for software. The principal developers of SEM are Robert Wallace and John Stockenberg. The

method has origins in Structured Analysis and Design, the Software Cost Reduction (SOR) project and Systems Analysis of Integrated Network Tasks (SAINT); it was first used in its complete form in 1981. Recent systems developed include a Trident Defensive Weapons System/Control Subsystem and a SAM Missile simulator.

System requirements are analysed using modeling techniques based on the U.S.A.F.'s Integrated Computer-Aided Manufacturing Definitional Methods and Structured Analysis. The SOR techniques are used to analyse specifications and to develop the architectural design of the software. Evaluations through simulation and prototyping are accomplished using SAINT. SEM also contains integration techniques that guide the user in bringing together the various products and results. A comprehensive view of the method has been presented in [Wall87].

3.6 STATEMATE

Visual formalisms are the basis for this method developed by A. Pnueli and D. Harel and first used for an avionics application in 1983. The method includes tools and techniques for dealing with specification, design and analysis of systems. Additional capabilities include management support tools and simulation for testing designs.

A conceptual model is used to deal with notions, entities and procedures that are relevant to reactive system development; i.e. responsive to both subsystems and environment. Three representational techniques are utilized to describe systems:

1. Statecharts, an extension of finite state diagrams, are used to specify behavior;
2. Activity-charts are used to specify system functionality;
3. Module-charts are used to specify system structure. Additionally, there are parts of the method for conducting "what if" analyses and developing prototypes. Further information about the method is provided in [Hare87] and [Hare88].

4. SURVEY QUESTION RESPONSES

In the survey used to gather data for the catalog, the authors solicited information about specific features of methods which can be identified prior to a method's use in the development process. Though not as significant as the effect that results from the use of

a method during the development effort, such features do represent information which clearly describes the method rather than its product.

Accordingly, typical of the questions asked in the survey were what development phases are addressed by the method, what type of programming practices, analysis and review techniques are espoused by the method, and what modes of representation are used by the method. Other questions focused on the specific ways that the method provides assistance to the developer in areas such as error detection, incorporation of change, and identification of reusable components. Information also was sought regarding automated support provided with the method, available training, and publically available bibliographic references about the method. In addition, the opinion of each developer was solicited concerning the amount of time, and the type of background needed to learn the method.

In the sections which follow, a selection of these questions are presented along with the responses received for the six methods.

4.1 Techniques Employed by the Method

In an effort to learn the degree of importance of well-known programming concepts and practices to proper use of the method, the respondents were asked to describe the extent to which certain concepts and practices were used in the method. In Table 1, below, the responses are coded as: *E* for essential, *C* for compatible, and *U* for unknown relationship to the method. No respondents answered that the listed practices were inconsistent with their respective methods.

TABLE 1 - Use of Programming Practices.

Concepts and Practices	METHOD					
	A	D	M	O	S	S
	D	C	A	O	E	T
	M	D	S	A	M	M
Stepwise Refinement	E	E	C	C	E	C
Information Hiding	E	E	E	E	E	C
Process Abstraction	E	E	C	E	E	C
Abstract Data-types	E	E	E	E	C	C
Structured Programming	E	E	C	U	C	C
Genericity	E	C	E	C	C	C
Inheritance	C	U	C	C	U	U
Use of assertions	C	C	C	C	C	C
Module coupling/cohesion	E	E	E	U	C	C

In an effort to learn what analysis and review techniques were essential to using the method, the respondents were asked to describe the extent to which each of certain well-known techniques were used in the method. The responses were to be coded with *R* for required, *E* for encouraged, and *N* for not addressed by the method. Table 2, below, shows the responses obtained to this question.

TABLE 2 - Use of Analysis and Review Techniques.

Analysis and Review Techniques	METHOD					
	A	D	M	O	S	S
	D	C	A	O	E	T
	M	D	S	A	M	M
Data-structured analysis	N	R	N	R	E	N
Data-flow analysis	R	R	R	R	R	E
Control-flow analysis	R	R	E	R	R	E
Decision tables	N	E	E	E	R	N
Formal proof techniques	N	N	N	N	N	E
Design reviews	R	R	R	E	E	E
Code walk-throughs	R	R	E	N	N	N
Change control review	R	E	E	N	N	N

4.2 Technology Transfer Issues

Another series of questions asked the respondents to estimate certain learning times, such as the time required to attain a high-level of understanding. It is also important to know how long it will take to learn enough to make reasonable use of a method and how long it might take to become an expert. The results obtained are summarized in Table 3.

TABLE 3 - Learning Times for Methods.

Estimated Learning Times	METHOD					
	A	D	M	O	S	S
	D	C	A	O	E	T
	M	D	S	A	M	M
Days - project manager overview	2	3	1	1	2	1
Days - experienced developer learning essentials	5	5	5	5	10	2
Months - to become an expert user	3	2	2	6	2-4	3

Developers of software development methods assume, perhaps implicitly, that the users of their methods are capable of understanding and following the instructions given. Consequently, one type of

precondition for proper use of any method is the set of assumptions made about the method user's education and experience. The questionnaire posed several questions about these preconditions; questions about technical or college-level education as well as about experience with programming languages and in previous development situations. In Table 4, the results are summarized; note that OOA is omitted because no answers were provided by the respondent.

TABLE 4 - Minimum Education and Experience.

Minimum Qualifications	METHOD				
	A	D	M	S	S
	D	C	A	E	T
	M	S	S	M	M
College-level technical education	4	4	0-1	4	4
Number of years of development experience	1-2	3-5	1-2	3-5	0
Number of prog. lang. (working knowledge)	1	2	1	1	1
Number of systems with which one has experience	1	3-4	2	1	1

Finally, it is useful to know the type of theoretical background which is expected of the developer in order that he or she can learn to use the method. The survey attempted to identify the necessary theoretic concepts in the question which follows.

Question: List the major theoretical construct(s) which should be understood by an experienced developer in order to successfully use the method.

The responses of the developers are summarized below.

- ADM** Abstract state machines, abstract data types, object abstraction, process recursion, and knowledge of the Ada language.
- DCDS** F_NETS (control-flow structure), and R_NETS (stimulus-response thread structure).
- MASC** Concepts of asynchronous concurrency, and data-flow analysis.
- OOA** Entity-Relationship-Attribute Modeling, data-flow diagrams, finite-state machines, object classes.

SEM The concepts which are typically found in an undergraduate computer science program.

STMT State machine knowledge.

5. ADDITIONAL RESPONSES

The survey also contained a series of questions designed to provide brief, but specific, information relative to certain features of methods. The questions focused on key issues such as client involvement, modes of representation, reusability, and assistance provided by the method relative to certain aspects of the development process. In the sections which follow, a sampling of these questions is provided, along with a paraphrasing of the responses provided for each of the methods.

For each section, a brief rationale is given for inclusion of the question in the survey. No comments, however, are offered on the substance of the responses. It is the authors' belief that the reader is best able to judge the appropriateness of the responses based upon his or her own experience in developing systems. Note that where responses are missing, information was not provided; further investigation is necessary to clarify these areas. The authors caution that the reader should avoid drawing any negative inference in the situation where no response is reported.

5.1 Client Involvement

A key component in the use of a method involves what aspects of the method assist in communicating with the client, and where the method requires the involvement of the client. Two questions dealing with this issue were included in the survey.

Question: What specific features or aspects of the method are designed to facilitate and coordinate communication between the client and the development organisation?

ADM Communication is facilitated through the use of graphics, and a verb/direct object program design language. The method's close relationship to the Ada language also facilitates communication for implementations which are Ada-based.

DCDS The method employs English-like languages to describe elements, relations and attributes. Extensive graphics are also used, and automated document generation is provided.

MASC The method specifically requires that the developer identify required functions and the required interactions with the environment. The design technique used enables traceability to be established back to these items.

OOA The several models used represent the system from several perspectives and at various levels of detail. The representations employed are highly graphical, precise, and unambiguous.

SEM The method employs client-oriented, high-level description techniques (data-flow, control-flow, and information modeling) to ease client review. The method also provides traceability of requirements to the final product.

STMT Because the system specifications can be executed and prototyped, the development team can demonstrate proof of analysis concepts in an animated execution or prototyped form. Furthermore, at any stage of project development, the method's prototype module can generate code from the specifications which can be run in the target environment, thus allowing for further evaluation by both developer and client.

Question: Specifically, what procedures does the method provide for involving the client during the software development process, and where does this involvement occur?

ADM By being recursive [sic] and developing the design and the code early, it provides many of the benefits of rapid prototyping without the disadvantages. Since the design is compilable, the client is able to review a tested design rather than mere paper design.

DCDS Automatic facilities are provided for producing documents for System Requirements Review, for System Design Review, and for the Software Specification Review. Additionally, preliminary and detailed versions of the SDD are produced for the Preliminary Design Review and the Critical Design Review. The method also supports the production of test reports for the Functional Configuration Audit and the Physical Configuration Audit.

MASC The client is involved in "signing off" the required functions and interactions at the beginning of the design stage. Additionally, the client must approve the acceptance test procedures.

OOA The method requires interviews with the client during the requirements gathering stage. The client also participates in the OOA document walkthroughs.

SEM The high-level graphical language used at the front end of system development is specifically designed to facilitate client review, and even enable client participation in system definition.

STMT At any phase in development, the client is able to review the executable specifications and/or the prototype code which is provided by the method.

5.2 Modes of Representation

Central to the use of a method are the modes of representation selected by the method to describe the evolving system. Providing a variety of modes of representation enhances the ability of the developer to analyse the system, and facilitates communication among the technical team, management, and the client. On the other hand, a key to the use of multiple representations is the ability to translate from one mode to another.

Question: Use of what modes of representation, either textual or iconographical, are required or strongly encouraged by the method?

ADM The method requires the use of Petri nets, data-flow diagrams, control-flow diagrams, hierarchy charts, Buhr diagrams, Firesmith diagrams, and a program design language. Use of finite-state diagrams and a formal specification language is strongly encouraged by the method.

DCDS The method requires the use of control-flow diagrams, flowcharts, narrative overviews of modules, a formal specification language, and specified documentation templates. The method strongly encourages, and provides automated support for, the use of data-flow diagrams, entity-relationship diagrams, decision tables, and a program design language.

MASC The method requires the use of a narrative overview of modules, data-flow diagrams, and hierarchy charts and other diagrams specific to the method. The use of finite-state diagrams, a program design language, structured English and specified documentation templates is strongly encouraged by the method.

OOA The method strongly encourages the use of finite-state diagrams, data-flow diagrams, entity-relationship diagrams, object communication diagrams, and object operation diagrams.

SEM The method requires the use of data-flow diagrams, control-flow diagrams, entity-relationship diagrams, decision tables, mathematical notation, and specified documentation templates. Additionally, the use of Petri nets is strongly encouraged.

STMT The method requires the use of finite-state diagrams, statecharts, activity-charts which are similar to data-flow diagrams, and module charts to show structure. The method strongly encourages the use of a formal specification language and specified documentation templates.

Question: If the method uses several modes of expression, identify the cases in which a mapping rule is prescribed which enables translating from one mode to another.

ADM 1) Data/control flow diagrams → subassembly object-oriented design diagrams;
2) object diagrams → data/control flow diagrams.

DCDS 1) F_Net (control flow) and text database information → IDEFo diagram (data-flow);
2) Logic flow diagram → a PDL; 3) Extended ERA graphics → HIPO charts.

OOA Guidelines are provided for mapping entity-relationship diagrams into Booch and Buhr diagrams.

SEM Data-flow diagrams → Petri nets;
2) data-flow diagrams → control-flow diagrams; 3) data-flow and control-flow diagrams → Template Based Specifications.

STMT 1) Activities → modules; 2) modules → activities charts; 3) control activities within the activities charts → statecharts.

5.3 Transformation Across Phases

In the two questions which follow, information is solicited as to how the method assists in moving from one phase of software development to the next, and what features in the method assist the developer in maintaining consistency among specification, design, and code.

Question: Specifically, how does the method facilitate the transformation across phases of the software process? (For example, from specification to design, or from design to code.)

ADM The method uses object-oriented data-flow diagrams which lead naturally to subassembly OOD diagrams. The Ada-oriented graphics and compilable PDL facilitate the transformation from design to code.

DCDS Transformation across phases is partially automated. Upon completion of the system requirements database, the elements to be forwarded are automatically written to a file. When the software requirements database is opened, this file provides the information whereby the elements from upstream can be transformed into elements of the downstream language. The same process is repeated when opening the distributed design phase, the module development phase, and the test phase. The languages provided for each phase are similar, though specific to the phase.

MASC The method achieves coherence between the design architecture and the implementation architecture by making them identical. Thus, no transformation is required.

OOA Object-classes which are identified through use of the method should map logically onto design classes, and these in turn should map in a one-one manner onto source code object classes, such as Ada packages.

SEM Detailed procedures are provided by the method which provide for transformations from each representation to the next.

STMT Transformation across phases is facilitated by supporting several conceptual levels with the use of the same model. The method provides the reports, executable models, related databases, and test suites to support this common model.

Question: Specifically, how does the method assist in ensuring that consistency is maintained among specification, design, or code when changes are made to any of these three entities?

ADM Maintaining consistency of requirements and design is facilitated because the method uses object-oriented data-flow diagrams which lead naturally into the subassembly object-oriented diagrams of

the design. Consistency of design and code is maintained through the use of Ada-oriented graphics and a compilable program design language which evolves into the deliverable code.

DCDS The method has rules and constructs for elements, relations, and attributes which assist in maintaining traceability from requirements to design, and from design to both code and tests.

MASC The method guarantees total consistency between the design structure and the implementation structure. Inconsistency cannot be introduced by implementors since any change must be introduced at the design level and then propagated into the implementation.

SEM All phases of development share a common semantic base, and thus traceability is built into the development process. There is never any ambiguity which will arise in tracing from high-level requirements to specifications and design.

STMT The method ensures consistency through the use of test scenarios. Test results are defined and achieved at different phases in the development process.

5.4 Other Development Issues

In this last section, responses are presented for questions which deal with the assistance provided by the method for the following:

1. Early detection of errors and inconsistencies;
2. Incorporation of changes in requirements;
3. Addressing timing constraints and concurrency;
4. Identification of reusable components.

Question: Specifically, how does the method assist in the early detection of inconsistencies and/or errors?

ADM By employing a recursive [sic] technique of "design a little, code a little, test a little", the method produces early compilable designs and early tested code.

DCDS The method provides automated facilities for checking completeness and consistency. A check is provided for use at the end of every major step.

MASC The complete design architecture is checked for self-consistency by a procedure known as "status progression". This procedure ensures that the design structure is sound before implementation is undertaken. A system can only achieve fully-checked status when all modules upon which the system depends have themselves achieved a similar status.

OOA Rules are provided for establishing the consistency among the entity-relationship model, the finite-state model, the data-flow diagrams, the object interaction/communication diagrams, and the object operation model (modified Booch diagrams).

SEM The method provides a simple, easily understood graphical language for defining system functions and behavior. The method also employs a built-in review procedure, and the use of simulation for dynamic feedback.

STMT The method provides a mechanism whereby a model of the system can be created, syntactically analysed, executed, dynamically tested, prototyped, and debugged.

Question: Specifically, how does the method assist in reducing the effort needed to fully incorporate changes in the requirements?

ADM As an object-oriented approach, the method produces software that is more extensible because changes to requirements are better localized in the design due to a lower level of data coupling.

DCDS Two element types are provided for assisting in incorporating change. The element type **CHANGE_REQUEST** is used for formal changes in baselined requirements. The element type **DECISION** is used for refinements that arise during development.

MASC The impact of changes can easily be traced to the affected areas of the design. The method's emphasis on well-defined interfaces and on the general architectural features yields a high level of decoupling within the design, thus limiting the effect of change.

OOA The method employs a partitioning of the requirements based upon object-classes. Requirements are stated exactly once, and are partitioned around other closely related requirements.

SEM The method's use of separation of concerns and levels of granularity ensure that for each proposed system change, there is only one place in the statement of requirements where the change needs to be incorporated. The ripple effect is avoided.

Question: What specific aspects of the method address requirements of the target system which involve timing constraints and/or concurrency issues?

ADM The method uses timing diagrams, and employs Petri nets, Buhr diagrams, and a task sequencing language to address concurrency issues.

DCDS For the software requirements, the method requires that completion conditions and delay events be addressed. During software design, timing constraints are specified for each routine. Assistance is provided for addressing concurrency through a full set of language constructs, and the availability of an analytical tool.

MASC Timing constraints are addressed in the behavioral model which encompasses concurrency, process synchronization, and deterministic scheduling. The design representation specifically distinguishes between concurrent (active) processes and passive data-objects.

OOA Concurrency is addressed by the fact that each instance of an object class is allowed to exist at any point in its behavioral lifecycle.

SEM Timing and resource constraints are analyzed through the use of control-flow (behavior) models. Simulation is used to further analyze system timing, concurrency issues, and resource contention.

STMT Timing constraints and concurrency issues are incorporated in the statechart modeling, and can be dynamically tested.

Note that a question involving spatial constraints was also asked on the survey. Unfortunately, the set of answers provided for the question were generally uninformative. In the authors' opinion, this might be explained by the tendency of software development methods to give too much of a "black-box" view to those concerns which involve hardware.

Question: Specifically, how does the method assist in identifying possible reusable components such as design or code?

ADM The method has an additional step in the design decision process where the issue of reuse is addressed. Reusable candidates naturally occur since the default packages implement abstract data types and abstract state machines.

DCDS The module development procedure provides guidelines for selecting modules which should be kept in the reusability library. This procedure also assists the developer in setting up and maintaining a reusability library using DCDS tools.

MASC The design architecture is predicated on the concept that all components are derived from templates. Thus, all templates are by their nature reusable, and are always specified in terms of the access interfaces that they provide or require.

SEM Emphasis on the use of separation of concerns and of information hiding assists in developing reusable specification and design components.

STMT The method encourages the identification of units that can be treated as reusable components.

6. OBSERVATIONS AND CONCLUSIONS

In the authors' opinion, there are several interesting trends which can be observed from the data presented. These include:

- The minimum qualifications generally needed by developers to use these methods includes a four-year undergraduate technical education and from one to five years of development experience.
- The use of data-flow analysis and finite state machines are central to these real-time methods.
- Iconographical representations are extensively used by these methods.
- Multiple representations of the evolving system are frequently employed, and the methods recognize the importance of providing rules or guidelines for mapping from one representation to another.

- While reusability is viewed as an important issue, specific rules for creating and/or identifying reusable components have not been completely addressed.
- Use of assertions and formal proof techniques are not currently an integral part of these methods.

It would be interesting to learn whether the same trends can be observed for other real-time development methods.

Based on their experience with this project, the authors conclude that hard real-time requirements have been addressed more fully by methods than spatial constraints associated with embedded systems. Finally, the authors remain convinced that the use of a uniform format for presenting information about methods provides a basis by which individual developers and organizations can judge what methods best suit their needs.

BIBLIOGRAPHY

- [Alfo87] M. Alford, "DCDS Multiple View Approach to Closing the Requirements/Design Gap", presentation at the Fourth Conference on Methodologies and Tools for Real-Time Systems, Washington, DC, Sept. 1987.
- [DCDS87] "Distributed Computing Design System: A Technical Overview", TRW System Development Division, Huntsville, AL, July, 1987.
- [Harel87] D. Harel, "Statecharts: a Visual Formalism for Complex Systems", *Science of Computer Programming*, Vol. 8, No. 3, June, 1987, pp. 231-274.
- [Harel88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtut-Trauring, "STATEMATE: a Working Environment for the Development of Complex Reactive Systems", *Proceedings of the Tenth International Conference on Software Engineering*, Computer Society Press, Washington, DC, 1988.
- [Maha87] L. Mahajan, M. Ginsberg, R. Pirchner, and R. Guilfoyle, "Software Methodology Catalog", Report No. C01 001 C8 0001, Center for Software Engineering, Advanced Software Technology, U.S. Army CECOM, Fort Monmouth, NJ., October 1987.
- [Smit88] M. K. Smith and S. R. Tockey, "An Integrated Approach to Software Requirements Definition Using Objects", *Proceedings of the Tenth Structured Development Forum*, San Francisco, CA, August, 1988.
- [Wall87] R. H. Wallace, J. E. Stockenberg, and R. N. Charette, *A Unified Methodology for Developing Systems*. Intertext Publications, New York: McGraw-Hill, 1987.

AUTHORS

Richard Guilfoyle is a Professor of Mathematics at Monmouth College. He has taught mathematics and computer science courses for over two decades and he consults in the areas of simulation, numerical analysis and programming languages. He received a PhD in Mathematics from Stevens Institute of Technology. He coauthored the two editions of the *Software Methodology Catalog* while engaged as a consultant for Teledyne Brown Engineering. He is a member of the ACM, SIGPLAN, SIGSIM, SIGSOFT, and of the IEEE Computer Society.

Richard Pirchner is an Associate Professor of Computer Science at Monmouth College. He received an MS in Mathematics from St. John's U. and an MS in Computer Science from Rutgers University. His areas of interest include software engineering, programming languages, database systems, and computer science education. He coauthored the two editions of the *Software Methodology Catalog* while engaged as a consultant for Teledyne Brown Engineering. He is a member of ACM, SIGADA, SIGMOD, SIGSOFT and the IEEE Computer Society.

Laurel Von Gerichten is a principal programmer analyst at the New Jersey office of Teledyne Brown Engineering. For the past two years, she has been task leader of Software Methodology Research, involving production of the *Software Methodology Catalog*, eds. 1 and 2, as well as inquiry in the area of method evaluation. She received a BA in English from the University of Chicago, an MS in Education from Northern Illinois U., and an MS in Computer Science from Mon-

mouth College. Her interests include software engineering, programming languages and linguistics. She is a member of ACM, SIGSOFT, and of the IEEE Computer Society.

Marilyn Ginsberg is a senior systems analyst at Teledyne Brown Engineering where she participated in the development of the Software Methodology Catalog and in research into method evaluation. She has also performed independent verification and validation of Government procured software, including Ada support systems. Her experience in software engineering includes applications development, software support, software development tools, testing standards, and software quality assurance. She previously worked for AT&T and Concurrent Computer Corp. She received a BA in Mathematics from Rutgers U., and is currently enrolled in the graduate computer science program at Monmouth College. She is a member of the ACM, SIGSOFT, SIGADA, SIGMETRICS, and of the IEEE Computer Society.

Donald Clarzon has over 20 years of experience with computer systems in military and computer vendor environments. He has been involved with the Ada programming language since 1981 when he served as a reviewer of the first definitive version of the Ada Language Reference Manual during the American National Standards Institute's standardization process. He is a Principal Systems Analyst at Teledyne Brown Engineering (TBE) where he has been involved in Ada analysis projects; he also is the principle designer and instructor of a series of Ada language courses offered to members of the DoD community through TBE. He holds a B.S. degree in Electrical Engineering and he is a member of SIGADA.

TECHNIQUES FOR OPTIMIZING ADA/ASSEMBLY LANGUAGE PROGRAM INTERFACES

Eric N. Schacht

Computer Sciences Corporation

Abstract.

This paper describes several techniques for interfacing assembly language routines to Ada programs. These techniques exhibit varying degrees of complexity, maintainability, and performance. The basic options and mechanics involved in establishing the interface between an Ada and assembly language program will be discussed. The ASMOG assembly language will be used for example purposes, although the interfacing techniques will be presented in a generalized fashion so that readers from a broad spectrum of assembly language product backgrounds may benefit. Increasingly sophisticated techniques for optimizing performance will then be given. Benchmark test results will highlight the trade-offs between program maintainability and performance efficiency. Special emphasis will be given to the parameter passing mechanism. The paper concludes with a set of guidelines for managing the implementation of Ada/assembly language program interfaces.

Background.

There are a number of reasons why the ability to interface lower level programs to an Ada program is important. A primary reason is simply to enable access to lower level machine capabilities. Another important reason is to improve program performance characteristics of certain speed critical program operations. The primary focus of this paper will be directed towards the latter concern.

The interfacing techniques described in this paper were developed during a research effort to analyze the possible Ada conversion of a major DoD weapon system employing fiber optic technology. Highlights of the results of this research were reported by the author at the Sixth National Conference on Ada Technology in a paper entitled "Research on the Ada Conversion of a Distributed, Fast Control Loop System". This paper identified and explained how several performance critical areas of the existing gunner station

software were converted from PL/M-86 to Ada and then benchmark tested. The benchmark test results were analyzed, and a list of compiler characteristics critical to a successful Ada conversion was developed - characteristics both within and beyond the current Ada language standard. Assembly language program interfaces were mentioned in this paper as a means to obtain access to vital lower level machine facilities such as the shared resource synchronization mechanism, and to improve the execution speed of performance sensitive areas. This paper extends the research results of the original paper by examining in detail the assembly language program interfacing techniques used.

Why assembly language interfaces are important.

Higher order languages allow us to work at more abstract levels than do more primitive machine level languages. This is vital to software developers for obvious reasons. Working at a higher level of abstraction results in substantially increased development productivity and enhanced ability to create understandable and maintainable programs.

Embedded computer systems, particularly those found in advanced weapons systems, present special challenges for software engineers. Such systems often require interfaces to numerous external devices or other external systems, and these interfaces are often of a specialized or non-standard nature. Raw processing power and memory size are often severely constrained because of the physical environment in which the embedded computer system must operate. As a result, programming efficiency can become very critical.

Because of the special environmental factors found in embedded computer systems, access to machine dependent features is important for two reasons. One reason is to support efficient external interfaces, the other is to improve the execution speed of the application software.

The Ada representation specifications are the vehicles for permitting us to reference low level features with Ada constructs that are used at a higher, more abstract level. These higher level constructs or entities are used in high level solutions at high levels of abstraction. Representation specifications allow us to map these high level constructs to the underlying machine.

In some time critical applications, an approach using Ada code only may fail to meet performance requirements. Sometimes a resort to assembly language code is the only means available to improve system performance to acceptable levels. Furthermore, some embedded systems applications require access to special machine dependent features which may simply be unavailable through high level Ada constructs and programming facilities, and these machine level features may be absolutely indispensable for the application to perform its intended function. This is especially true with Ada compilers which lack the full range of Chapter 13 facilities. For example, consider an embedded application with multiple central processing units that execute processes which communicate with one another through data structures in shared memory over a shared system bus. The pragma SHARED specifies that a read or write operation on a shared variable must be implemented as an indivisible operation. This pragma enables a variable shared by different concurrently running processes to be accessed and updated by those processes without risk of corruption due to simultaneous access. If the Ada compiler used in generating the applications software does not support the pragma SHARED, then a resort to a lower level machine language is the only means to implement a shared data structure.

Ada provides the means to write low level machine statements without stepping outside of the language itself through the use of representation specifications. This is ideally done by creating a package that exports a record abstracting the processor's instruction set mnemonics. Access to low level machine facilities is also possible through pragma INTERFACE. The arguments for this pragma are a subprogram name and the name of the language (such as an assembly language) in which the subprogram was written. Pragma INTERFACE therefore is used to specify to an Ada compiler that an object module built in another language is to be incorporated for some specified subprogram. This approach involves stepping outside of the Ada language itself, and is obviously less desirable than the full Ada level approach of abstracting the processor's instruction set. Still, with some Ada compilers this

approach may be the only avenue available to capitalize on benefits obtainable from low level machine features. With some time critical applications, this may mean the difference between meeting and not meeting performance requirements.

Motivation for technique development.

A highly time critical module in the PL/M-86 based gunner station software interfaces with an autotracker. Sixty times a second, this device interrupts the cpu to provide a value for the missile positional error with respect to a moving target on which the autotracker is fixed. Also, sixty times a second, this error value must be processed by the digital autopilot software running on a different cpu. The digital autopilot program uses this error value to derive a fin movement command. The fin command generated is designed to move the missile so that future error signals will be nulled.

It is obvious that this application is time critical. The program which processes the autotracker data contains numerous equations which perform shift and rotate and bit-wise Boolean operations. In an ideal PL/M-86 to Ada code conversion, the use of representation specifications would be the preferred method of obtaining the required values from incoming autotracker data. This would be the approach most consistent with sound software engineering principles. The Ada compiler used in this research did not offer the record type specification, but did offer an interface to ASM86 assembly language. But even if the record type representation specification is provided by an Ada compiler, there is no guarantee that the object code generated by the compiler would be efficient enough to meet performance requirements. If this is the case, the software engineer has no alternative other than an assembly language interface in order to satisfy performance requirements.

In the following section, various techniques are presented which were developed to provide the functionality of bit shift and rotate and bit-wise Boolean operations required by the autotracker software. The techniques will be presented in decreasing order in terms of soundness and desirability from a software engineering standpoint. As the reader progresses through these techniques, several generalities become obvious. One is that as performance efficiency improves, there is a corresponding loss in abstraction. As the abstraction quality declines, there is a corresponding decline in program understandability and maintainability. Generally speaking, performance gains are realized by compromising principles of "good"

software engineering.

The primary intent for presenting these techniques is not to specifically show how to optimize bit shift and rotate and bit-wise logical operations, but rather to demonstrate the abstraction quality versus performance trade-off involved as programming approaches seek greater and greater levels of performance efficiency. Another prime intent is to explain some mechanisms available for optimizing time sensitive program operations. From this experience, a set of guidelines will be developed to aid the software engineer faced with challenging functional and performance requirements which can only be met by stepping outside the Ada language. These guidelines can serve as a simple decision model when making decisions concerning the abstraction quality versus performance trade-off.

Technique # 1 - "Pure" Ada.

The first approach involved constructing an Ada package which provides the equivalents to the PL/M-86 functions: SHR (shift right), SHL (shift left), SAR (shift arithmetic right), SAL (shift arithmetic left), ROR (rotate right), and ROL (rotate left). Bit-wise Boolean operations were supplied by "withing" a compiler specific package called "UNSIGNED". These functions operate by analyzing on a bit by bit basis each bit in an operand passed. The resulting value is set on a bit by bit basis. These tasks are performed using a bit mask array and the bit-wise Boolean operations from the compiler specific package. This package is listed in Figure 1.

These functions were benchmark tested against their PL/M-86 counterparts. The results showed that this approach is horrendously inefficient. Some of the test results are included at the end of Figure 1. All benchmark tests conducted during this research were run on a Zenith 248 personal computer running at 8 Mhz. The reasons for this extreme inefficiency will become clearer as subsequent techniques are analyzed.

Technique # 2 - "Twe" Ada with constraint checking deactivated.

The second approach involved a rewrite of the previous algorithms to provide the same shift and rotate functions through multiplication and division by some power of two. Again, bit-wise Boolean operations were performed by the compiler specific package UNSIGNED. For this approach to work, constraint checking instruction generation was suppressed at

compile time. This technique represents the first compromise made with "good" software engineering principles. The compromise is that program reliability is reduced due to loss of constraint violation detection during runtime. The program code is given in Figure 2.

The performance of this set of algorithms improved significantly versus the first approach, but these functions were still much slower than their PL/M-86 counterparts. Some of the test results are listed at the end of Figure 2.

From these test results it can be seen that the speed of the Ada programs is a function of the number of bits shifted or rotated. This is true because the processor handles multiplication as repeated addition and division as repeated subtraction.

Technique # 3 - Interfaceably language routines which perform shift and rotate and bit-wise logical operations.

It became apparent with the experience of the second approach that a step outside the Ada language would be necessary to approximate the performance of the PL/M-86 functions. The third approach was to analyze the effect of using assembly language routines to perform the shift and rotate functions. The assembly language routines built were patterned after the code seen in the assembly language dump of the PL/M-86 benchmark programs using these functions.

One of these assembly language programs is shown in Figure 3 - the SHL function. The parameters are the bit pattern to be shifted and the number of bits to be shifted. The corresponding Ada program function declaration statements which establish the link to the assembly language routine are also included at the end of Figure 3. These statements include the function declaration, the pragma INTERFACE, which instructs the compiler that an object module is to be supplied for the corresponding function name, and a compiler specific pragma called pragma INTERFACENAME which completes the declaration. Note how the function parameters must be described in the assembly routine as some offset from an address contained in the BP (base pointer) register.

Benchmark programs were then written in Ada and PL/M-86 which contain numerous equations from the autotracker software involving bit shift and rotate and bit-wise Boolean operations. The Ada program's bit-wise Boolean operations were performed by the compiler specific package.

Comparisons of the complete assembly language dumps of the Ada and PL/M-86 programs provided some interesting insights which would explain the results of subsequent benchmark tests.

The most obvious concern which arose was due to the greater number of instructions generated in the Ada program because of the overhead of transferring control to and returning from an interfaced procedure. This overhead was incurred for each and every bit shift and rotate and bit-wise logical operation. The PL/M-86 object code did not have this overhead - these operations were simply embedded in the normal instruction stream.

When further studying the details of the mechanisms involved in interfacing subprograms, some inefficiencies in the Ada object code were noted.

The user's manual for the Ada compiler stated that all interfaced subprograms are called by an individual FAR call through a pointer allocated in the global data area. The implication of this is significant - it means that the interfaced subprogram has a different code segment value than the calling program. For the procedures in the PL/M-86 benchmark program, both the calling routine and the called routine have the same value in the code segment register.

It is important to understand this mechanism of subprogram interfacing because of its effect on execution speed. In a call to and return from a procedure, the control transfer instructions (JMP, CALL, RET, etc.) break the current instruction sequence and cause program execution to resume elsewhere in the program code. The assembler uses the procedure type label (NEAR or FAR) to determine whether to produce an opcode that changes only IP (the instruction pointer register), or an opcode that changes both CS (code segment register) and IP.

In an ASM86 assembly language program, the type of a procedure (NEAR or FAR) is indicated to the right of the keyword PROC. The type associated with a procedure is used by the assembler in determining which CALL instruction to generate for the procedure. If a FAR is indicated, the long form of the CALL instruction is used. In this case, both the CS and IP values are changed when control is transferred, so a two-word return address (CS and IP) is pushed on the stack. For a NEAR procedure, only the IP gets changed, so the return address is a single word indicating an IP value. Since there are two kinds of return addresses, there are two kinds of RET instructions. The RET for a FAR procedure restores both CS and IP us-

ing values from the stack, while the RET used for NEAR procedures reloads only IP with the word stored in the stack. Both types of return instructions are specified with the mnemonic RET - the assembler automatically decides the appropriate one to generate.

The impact of the procedure interfacing overhead on the benchmark test results was substantial - the Ada program was 45% slower than its PL/M-86 counterpart. While the performance difference has substantially narrowed when compared to the second approach, this difference was still not narrow enough for the Ada version to satisfy performance requirements.

Technique # 4 - Narrowing the performance gap by minimizing the number of assembly language program interfaces.

Since the subprogram interfacing overhead is a constraint which could not be removed, the only option available for further performance improvement was to reduce the amount or frequency of calls to the interfaced assembly routines. Closer analysis of the autotracker software showed that most of the equations using logical and shift and rotate operators were concentrated in two areas of the program.

The fourth approach involved moving those parts of the equations which performed these operations to two interfaced assembly language routines - each routine corresponding to one of the program sections where these operations were clustered.

Minimizing the amount of the interfaced procedure call and return overhead did successfully narrow the performance gap. The Ada benchmark program was now only 15% slower than the PL/M-86 version. All involved Ada program variables were passed to the assembly language programs, along with the addresses of each of the variables. It was necessary to pass the addresses of these variables in order for the assembly language program to store calculation results in the appropriate Ada variable memory location.

This approach represents a severe and undesirable compromise with effective software design principles. In effect, the abstraction of the problem solution virtually disappears from the Ada program. It is shifted to the assembly language program, where the problem solution is at a greatly reduced level of abstraction. The assembly language code is naturally far more difficult to understand and maintain. The interface to the assembly language routine is very messy - each affected variable and its address (obtained via the

ADDRESS attribute) must be passed.

This interface is all the programmer sees when viewing the Ada code - there is not the slightest hint of what the program is dealing with or attempting to solve. The program can now be maintained with the aid of extremely lucid and verbose documentation, and an expert assembly language programmer. This approach obviously has undesirable ramifications from the program manager's standpoint.

With the performance differential narrowed to 15%, this is the first technique which becomes feasible given the imposed hardware and software constraints. However, the program management was interested in seeing if the performance gap could be narrowed even further.

Technique # 5 - Reducing the size of the parameter list.

Analysis of the object code resulting from the previous technique showed a tremendous amount of overhead involved in the call to and return from the two interfaced assembly routines. Every parameter passed in the function call had to be "pushed" on the parameter stack. Likewise, every parameter had to be "popped" from the parameter stack on return from the interfaced routine. The parameter list was sizeable for each routine - each affected variable and its address were loaded on and removed from the parameter stack.

The final technique sought to reduce this overhead by reducing the size of the parameter list itself. This was accomplished by placing all of the variables to be passed to the assembly language routine in a single record type object in the Ada program, and then passing only the starting address of this record to the interfaced routine. This eliminated all of the parameter "push" and "pop" activity associated with the procedure call and return, except for the starting address of the Ada record which in effect contained the parameter list.

Naturally, the assembly language routine had to be built with the knowledge of the exact structure of the Ada record containing the parameter list. First, the exact physical layout of the components within the record had to be determined. A special program was written which used the POSITION attribute to dissect and determine the exact physical layout of the record components. The POSITION attribute provides the offset of a particular record component with respect to the first of the storage units within the record. With this structural information known, the assembly language routine could be written

to address the locations of the Ada record components, given the starting address of the record as input.

Benchmark tests showed that the Ada vs. PL/M-86 performance gap had narrowed to less than 6%. This was deemed an acceptable performance level.

This technique probably represents the most powerful way to overcome performance bottlenecks short of rewriting the compiler itself. Unfortunately, a reasonable level of program understandability and maintainability has completely vanished here. This technique adds a new level of complexity over the fourth technique. It is also less maintainable. If any variable or equation at the source level changes, the assembly routine must be rewritten. Any change to a record component(s) within the parameter list embedded in the Ada record type object would upset the "hard-coded" knowledge of the physical arrangement of Ada components within the record. If an Ada application had many such performance bottlenecks which could only be solved using this kind of approach, one might legitimately question the merits of moving to a radically different solution such as a new compiler and/or hardware architecture. This can sometimes be considered a radical solution for a complex, embedded military application nearing maturity.

Lessons learned.

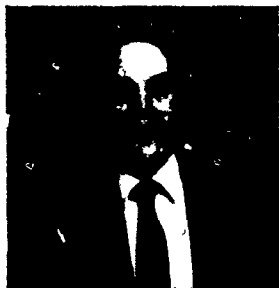
The order in which the techniques were presented progressed from higher to lower (or nonexistent) levels of abstraction, from lesser to greater compromises with sound software engineering principles, and from higher to lower levels of maintainability and understandability, and conversely, from lower to higher levels of performance efficiency.

There are specific lessons in this paper for the software engineer trying to overcome functional or performance bottlenecks encountered at the "pure" Ada language level, and lessons of a general nature for engineering management responsible for providing direction on how such technical problems are to be managed.

In solving a performance or functional bottleneck by resorting to a step outside the Ada language, a list of possible approaches should be generated. This list should be ordered in terms of desirability from a soundness of design, reliability, and maintainability standpoint. Benchmark programs should be developed and analyzed in terms of ability to meet performance requirements. The approach selected to solve the bottleneck should not necessar-

ily be the most performance efficient. Generally, the approach selected should be the one involving the highest level of abstraction quality, understandability, and maintainability, while still meeting performance requirements. Using this guideline as a decision model should help to minimize total system life cycle costs.

On any sizable embedded systems development program, it is important for systems engineering management to retain a library of techniques available to overcome performance and functional bottlenecks. Systems engineering should also provide direction on how to select and manage the implementation of these techniques. Documentation standards should become more rigid and require more time and effort for the lowest level approaches. Without such standards, the performance bottleneck solved today could become tomorrow's maintenance nightmare.



Eric N. Schacht is a Senior Computer Scientist in the Defense Systems Division of Computer Sciences Corporation in Huntsville, Alabama. His main areas of interest are CASE tools, systems engineering, Ada technology management, and large scale project management.

Schacht received his BS in management science from the University of Alabama in Huntsville in 1976 and MS in computer science from DePaul University in 1983. He is currently pursuing a PhD in computer science at the University of Alabama in Huntsville.

Schacht's address is Computer Sciences Corporation, Defense Systems Division, 200 Sparkman Drive, Huntsville, AL 35805.

with UNSIGNED, use UNSIGNED. -- imports bit-wise logical operations

package SHIFT_AND_ROTATE_FUNCTIONS is

```
START : TEMP : WORD;
BIT_MASK : array(WORD range 0..15) of WORD :=
  (16000016, 16000020, 16000016, 16000016,
   16000104, 16000108, 16000104, 16000104,
   16001008, 16001012, 16001008, 16001008,
   16001808, 16001812, 16001808, 16001808,
   16010008, 16010012, 16010008, 16010008);
```

```
function SHL(PATTERN : WORD ; COUNT : WORD) return WORD;
function SHR(PATTERN : WORD ; COUNT : WORD) return WORD;
function SAL(PATTERN : WORD ; COUNT : WORD) return WORD;
function SAR(PATTERN : WORD ; COUNT : WORD) return WORD;
function ROL(PATTERN : WORD ; COUNT : WORD) return WORD;
function ROR(PATTERN : WORD ; COUNT : WORD) return WORD;
```

end SHIFT_AND_ROTATE_FUNCTIONS;

package body SHIFT_AND_ROTATE_FUNCTIONS is

function SHL(PATTERN : WORD ; COUNT : WORD) return WORD is

begin

```
TEMP := 0;
START := 15 - COUNT;
```

```
for I in reverse 0..START loop
  if ((PATTERN and BIT_MASK(I)) > 0) then
    TEMP := TEMP or BIT_MASK(I - COUNT);
  end if;
end loop;
```

```
--TEMP := PATTERN * (2 ** CNT);      *** ALTERNATIVE EQUATION ***
```

return TEMP;

end SHL;

function SHR(PATTERN : WORD ; COUNT : WORD) return WORD is

begin

```
TEMP := 0;
```

```
for I in COUNT..15 loop
  if ((PATTERN and BIT_MASK(I)) > 0) then
    TEMP := TEMP or BIT_MASK(I - COUNT);
  end if;
end loop;
```

return TEMP;

end SHR;

function SAL(PATTERN : WORD ; COUNT : WORD) return WORD is

begin

```
TEMP := 0;
START := 15 - COUNT;
```

```
for I in reverse 0..START loop
  if ((PATTERN and BIT_MASK(I)) > 0) then
    TEMP := TEMP or BIT_MASK(I - COUNT);
  end if;
end loop;
```

return TEMP;

end SAL;

function SAR(PATTERN : WORD ; COUNT : WORD) return WORD is

begin

```
TEMP := 0;
```

```
for I in COUNT..15 loop
  if ((PATTERN and BIT_MASK(I)) > 0) then
    TEMP := TEMP or BIT_MASK(I - COUNT);
  end if;
end loop;
```

```
if ((PATTERN and BIT_MASK(15)) > 0) then
  START := 15 - COUNT - 1;
  for I in START..14 loop
    TEMP := TEMP or BIT_MASK(I);
  end loop;
end if;
```

return TEMP;

end SAR;

function ROL(PATTERN : WORD ; COUNT : WORD) return WORD is

begin

```
TEMP := 0;
START := 15 - COUNT;
```

```
for I in reverse 0..START loop
  if ((PATTERN and BIT_MASK(I)) > 0) then
    TEMP := TEMP or BIT_MASK(I - COUNT);
  end if;
end loop;
```

```
for I in 1..COUNT loop
  if ((PATTERN and BIT_MASK(START - I)) > 0) then
    TEMP := TEMP or BIT_MASK(I - 1);
  end if;
end loop;
```

return TEMP;

end ROL;

function ROR(PATTERN : WORD ; COUNT : WORD) return WORD is

begin

```
TEMP := 0;
```

```
for I in COUNT..15 loop
  if ((PATTERN and BIT_MASK(I)) > 0) then
    TEMP := TEMP or BIT_MASK(I - COUNT);
  end if;
end loop;
```

```
START := COUNT - 1;
```

```
for I in 0..START loop
  if ((PATTERN and BIT_MASK(START - I)) > 0) then
    TEMP := TEMP or BIT_MASK(15 - I);
  end if;
end loop;
```

return TEMP;

end ROR;

end SHIFT_AND_ROTATE_FUNCTIONS;

Shift and Rotate Function Test - 300,000 Iterations (PL/M vs. Ada Using Bit Masking Algorithms)

Operation	Pattern(operand)	Count(# bits moved)	Ada(seconds)	PL/M(seconds)
SHL	65,535	8	123	6
ROR	255	8	214	6

Figure 1. Program code for technique # 1
and benchmark test results

```

with UNSIGNED, use UNSIGNED; -- imports bit-wise logical operations
package SHIFT_AND_ROTATE_FUNCTIONS is
  START, TEMP : WORD;
  SAR_MASK : array(INTEGER range 0..15) of WORD :=
    (2#1000_0000_0000_0000#,
     2#1100_0000_0000_0000#,
     2#1110_0000_0000_0000#,
     2#1111_0000_0000_0000#,
     2#1111_1000_0000_0000#,
     2#1111_1100_0000_0000#,
     2#1111_1110_0000_0000#,
     2#1111_1111_0000_0000#,
     2#1111_1111_1000_0000#,
     2#1111_1111_1100_0000#,
     2#1111_1111_1110_0000#,
     2#1111_1111_1111_0000#,
     2#1111_1111_1111_1000#,
     2#1111_1111_1111_1100#,
     2#1111_1111_1111_1110#,
     2#1111_1111_1111_1111#);

  function SHL(PATTERN : WORD ; COUNT : INTEGER) return WORD;
  function SHR(PATTERN : WORD ; COUNT : INTEGER) return WORD;
  function SAL(PATTERN : WORD ; COUNT : INTEGER) return WORD;
  function SAR(PATTERN : WORD ; COUNT : INTEGER) return WORD;
  function ROL(PATTERN : WORD ; COUNT : INTEGER) return WORD;
  function ROR(PATTERN : WORD ; COUNT : INTEGER) return WORD;

end SHIFT_AND_ROTATE_FUNCTIONS;

package body SHIFT_AND_ROTATE_FUNCTIONS is

  function SHL(PATTERN : WORD ; COUNT : INTEGER) return WORD is
  begin
    TEMP := PATTERN * (2 ** COUNT);
    return TEMP;
  end SHL;

  function SHR(PATTERN : WORD ; COUNT : INTEGER) return WORD is
  begin
    TEMP := PATTERN/(2 ** COUNT);
    return TEMP;
  end SHR;

  function SAL(PATTERN : WORD ; COUNT : INTEGER) return WORD is
  begin
    TEMP := PATTERN * (2 ** COUNT);
    return TEMP;
  end SAL;

  function SAR(PATTERN : WORD ; COUNT : INTEGER) return WORD is
  begin
    TEMP := PATTERN/(2 ** COUNT);
    if (PATTERN and SAR_MASK(0)) > 0 then
      TEMP := TEMP or SAR_MASK(COUNT);
    end if;
    return TEMP;
  end SAR;

  function ROL(PATTERN : WORD ; COUNT : INTEGER) return WORD is
  begin
    TEMP := (PATTERN * (2 ** COUNT)) or (PATTERN / (2 ** (16 - COUNT)));
    return TEMP;
  end ROL;

  function ROR(PATTERN : WORD ; COUNT : INTEGER) return WORD is
  begin
    TEMP := (PATTERN/(2 ** COUNT)) or (PATTERN*(2**(16-COUNT)));
    return TEMP;
  end ROR;

end SHIFT_AND_ROTATE_FUNCTIONS;

```

Shift and Rotate Function Test - 300,000 Iterations
(PL/M vs. Ada Using Multiplication/Division Algorithms)

Operation	Pattern(operand)	Count(# bits moved)	Ada(seconds)	PL/M(seconds)
SHR	65,535	8	34	6
SHL	255	1	14	6
SHL	1	15	52	6
SHL	1	8	34	6
SAL	255	1	14	6
SAR	65,535	8	41	6
ROL	255	8	65	6
ROR	255	8	65	6

Figure 2. Program code for technique # 2
and benchmark test results.

```

NAME      SHL_PKG
SHL_CODE  SEGMENT BYTE PUBLIC
          ASSUME CS:SHL_CODE

```

```

; The following defines how arguments will be accessed.
; (Compare with the Ada DETAIL messages in the compiler listing.)

```

```

PAT      EQU      [BP+6]
CNT      EQU      [BP+8]

LEFTSH   PUBLIC    LEFTSH
PROC     FAR

          PUSH     BP           ; Establish frame for addressability to
          MOV      BP,SP       ; parameters from interfaced subprogram

          MOV      AX,PAT      ; Move value to be shifted into word register AX
          MOV      CL,CNT      ; Move number of bits for value to be shifted
                                ; into byte register CL
          SHL      AX,CL       ; Perform shift

;        MOV      SP,BP       ; Destroy the frame
          POP      BP
          RET      4           ; Return to Ada caller, popping IN parameters
LEFTSH   ENDP
SHL_CODE ENDS
END

```

```

function SHL(PAT : WORD ; CNT : BYTE) return WORD;
pragma INTERFACE(ASSEMBLER,SHL);
pragma INTERFACE_NAME("SHL","LEFTSH");

```

Figure 3. A sample assembly language program interfaced to an Ada benchmark program to provide the shift left function, and the associated Ada program function declaration statements.

System Simulation in Ada for the Project Manager

Kevin J. Cogan

Electronics Technology and Devices Laboratory
Ft. Monmouth, New Jersey 07703

Philip W. Caverly and Carmine Marino
Jersey City State College
Jersey City, New Jersey 07305

ABSTRACT

Ada can serve as a highly graphic, interactive and expressive language for the project manager. When equipped with a graphics package and maximum use of separate compilation units, Ada can be used to simulate future systems in order to provide necessary feedback to modify the statement of work early in the design phase to help ensure first pass success of delivered hardware. The simulated system can lead to the exploitation of common hardware platforms through system reconfiguration under software control.

statement of work. It functions at a high-level of abstraction which can be used by the PM as well as by soldiers who will utilize the equipment. It is postulated that a system simulation in Ada provides a cheaper and faster design assessment for use by the project manager than obtainable from a hardware prototype.

Ada vs. VHDL

Hardware description languages (HDLs) have been available for many years but there has been no clear standard until the DoD Very High Speed Integrated Circuit (VHSIC) program spawned the development of the VHSIC HDL (VHDL), now IEEE Standard 1076. VHDL provides the capability for describing the behavior, data flow, and structural models of digital circuits, but it is neither interactive nor graphic at the system level.

Despite the tremendous impact that VHDL is having on the electronics industry, VHDL in its present form does not provide an interactive simulation interface for user testing and executive decision making. In contrast, and with an appropriate graphics package, Ada can simulate the behavior of a system in a user-friendly and interactive environment providing useful feedback to the PM. The suitability of using Ada for lower level simulation of digital signal processing systems is presented by [Happel and Petrasko].

THE METHODOLOGICAL MODEL

Creating the software mockup begins at a high level of abstraction. The man-machine interface is decomposed into two separate units - the man and the machine. In the Ada paradigm, both can be viewed as tasks. The man is an active or pure user task which from time to time makes use (calls an entry) of the machine which, in the simplest case, is a passive or pure server task. The machine will accept

INTRODUCTION

Risk, budget and time are critical elements in project management. This is particularly true for future military systems. Military projects begin with a Required Operational Capability (ROC). The ROC is developed through an evaluation of the battlefield commander's needs and the threat analysis. A project manager (PM) is then selected to transform the ROC into a statement of work (SOW).

Usually the first tangible feedback from the SOW is a hardware prototype of the system developed under contract. The SOW must be precise, lest the prototype will not conform to the ROC. The cost due to changes in the SOW are directly proportional to the elapsed time in the project cycle. Consequently, changes early in the design phase are less costly than at the time of delivery of a working prototype. Costs due to changes in the SOW beyond this phase typically rise exponentially. Design and procedural anomalies revealed in operational tests may indicate preferred changes to the SOW, but the impact on budget and time often prohibits this course of action.

This paper will examine the use of a "software mockup" for a candidate system under development. A software mockup is defined as an interactive and graphic computer simulation derived from the

valid entry calls. When modeled correctly the machine task will react to human input according to its functional specification as given in the SOW. The synchronization of the man with the machine is modeled as a rendezvous between the calling and called tasks in Ada. Usually a change in the machine's state will occur as a result of the rendezvous which can be represented graphically on a CRT.

A graphic mockup driven by Ada software can be used as a valuable tool to assess response time, training requirements, incorrect steps in procedure, and alterations to the operator's manual. The feedback to the PM, through visual and hands-on interaction with or without soldier testing, helps to gather valuable insight into how well the SOW was interpreted. Although the need for preciseness in the SOW is not diminished by this method, changes to the software at this stage in the system development are far less costly and faster than if changes would have to be made after delivery of hardware. In the final analysis, recompilation of software is more acceptable than reconfiguration of hardware. Maximum use of separate compilation in Ada adds greater efficiency to the recompilation process.

VALIDATING THE METHODOLOGICAL MODEL

A system in the test phase of development was chosen to validate the methodological model. Known as the Water Quality Analysis Unit (WQAU), its purpose was to sample the purity of tactical water supplies produced by the currently fielded Reverse Osmosis Processing Unit. Presently, such testing is performed using wet chemistry kits which would be replaced by the electronic WQAU. An appropriate SOW was generated by a PM for the proponent Army command.

A prototype was designed and built under contract as shown in Figure 1.

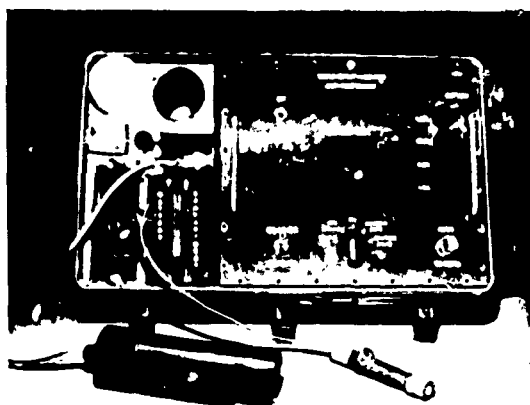


Figure 1. The delivered WQAU prototype.

The unit did not pass its operational field test. It failed the drop test, was 12 pounds overweight, and did not give consistent readings. Soldier interface difficulties were also revealed during the test. For example, the user manual expressly states that the WQAU must be placed in the CALIBRATE mode when initially turning on the power. This was easily defeated or ignored by leaving the mechanical toggle switch in the MEASURE mode.

Five attributes of water had to be measured - pH, total dissolved solids, temperature, turbidity, and percent chlorine. Each attribute function was selected by a five position switch. An analog probe provided current and voltage levels to the WQAU which was then processed and displayed digitally. To conserve battery power, a five digit read-out was required to display for 15 seconds and then go off. This requirement, although correctly engineered, did not take human factors into account. As would later be demonstrated, a software simulation would have flagged the latter two faults before developing hardware as well as given insight to an alternative hardware platform leading to a lighter weight package.

SIMULATOR DEVELOPMENT

The Machine as a Task

As stated earlier, the man and machine can be modeled as tasks to request and accept services. The calling task is man; the called task is the WQAU. Thus, any switch position that can be set by the human task is modeled as an entry into the WQAU. Changing the switch position effects the machine depending on the condition of its present state. Example: an entry to READ the pH of water would be accepted but have no effect if the power switch is OFF. Changes in state are initiated from a keyboard and a graphic response is produced on a CRT (Figure 2.)

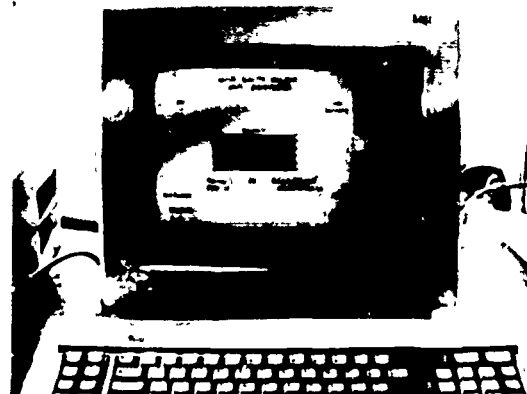


Figure 2. The WQAU Ada simulator on a PC.

Task MACHINE is represented as follows:

```
task MACHINE is
  entry READ;
  entry ON;
  entry OFF;
  entry MEASURE;
  entry CALIBRATE;
  entry PK;
  entry TURBIDITY;
  entry CNLORINE;
  entry TDS;
  entry TEMPERATURE;
end MACHINE;
```

Further discussion on the appropriateness of the task model will not continue here except to say that entries without parameters correctly convey the physical meaning of a switch - a rendezvous takes place, but nothing physically passes from human to machine. The machine merely reacts to the stimulus that a specific contact has been made.

An Ada Graphics Package

In order to provide a friendly interactive graphics environment to portray the WQAU in software, it was necessary to build a graphics package. As widely known, the Ada package STANDARD is rudimentary, and provides only primitive types and operations. Graphics is not part of the language, but Ada language designers expected that packages would be written by software developers as required and made available for later reuse. For this project, a package SET_LOCATION was initially developed in monochrome on a superminicomputer and later extended to incorporate color for a PC host. This package provided a tool to develop a likeness of the WQAU on a CRT by multiple selection of a row/column and a color from procedures in the graphics package when called with a specified character from procedure PRINT_SCREEN. The package specification for SET_LOCATION is shown in Figure 3.

```
package SET_LOCATION is
  USE ASCII;
  X,Y : INTEGER;
  TEXT : STRING(1..80);
  CLR : constant STRING:=
    ESC & "[2J" & ESC & "[1;1H";
  procedure MOVE (X,Y : INTEGER);
  procedure set_normal;
  procedure REV;
  procedure BOLD;
  procedure UNDERLINE;
  procedure BLINK;
  procedure NO_REV;
  procedure NO_BOLD;
  procedure NO_UNDERLINE;
  procedure NO_BLINK;
```

```
procedure SET_WIDTH_80;
procedure SET_WIDTH_132;
procedure invisible;
procedure black_f;
procedure red_f;
procedure green_f;
procedure yellow_f;
procedure blue_f;
procedure magenta_f;
procedure cyan_f;
procedure white_f;
procedure black_b;
procedure red_b;
procedure green_b;
procedure yellow_b;
procedure blue_b;
procedure magenta_b;
procedure cyan_b;
procedure white_b;
end SET_LOCATION;
```

Figure 3. Package Set_Location

The execution of each procedure in SET_LOCATION outputs an appropriate escape sequence derived from the terminal technical manual. For example, to take advantage of a 132 column terminal width, "procedure SET_WIDTH_132" is selected. The procedure body is hidden in package body SET_LOCATION as

```
procedure SET_WIDTH_132 is
  begin
    put(ESC & "[?3h");
  end;
```

With a sufficient graphics capability now in place, a bit of artistry was applied to adequately represent the WQAU on the CRT (Figure 2) through the execution of procedure PRINT_SCREEN declared in package WQAU. A help command was implemented to assist the user with allowable interactive entry calls to task MACHINE. Thus through the graphics interface, any user operation that could be performed on the hardware prototype could be commanded from the keyboard and the simulated result would be displayed. Naturally, only allowable commands had any effect. Any illegal commands or input errors had a null effect through judicious use of exception handling. Software development was accelerated by using separately compiled modules developed in the top-down design. A partial outline of the WQAU simulator in Ada is shown in Figure 4.

```
with SET_LOC,TEXT_IO; use SET_LOC,TEXT_IO;
package WQAU is
```

```
  type USER_ACTION is (TAKE_OUT,SOL_1,
                        SOL_2,SOL_3,READ,
                        ON,OFF,PM,TURBIDITY,
                        CHLORINE,TDS,TEMP,
                        MEASURE,CALIBRATE,HELP);
```

```
  subtype M_OR_C is USER_ACTION range
                        MEASURE..CALIBRATE;
```

```
  type STATUS is -- machine initial state
                  -- probe not in water
```

```
  record
    TYPE_OF_MEASURE
      : STRING(1..10):="PM          ";
    POWER : BOOLEAN := FALSE;
    MEAS_OR_CALIB
      : M_OR_C_TYPE :=CALIBRATE;
    SOLUTION : STRING(1..9):="AIR.DAT";
  end record;
```

```
  CURRENT_STATUS : STATUS;
  COND           : USER_ACTION;
  READY          : BOOLEAN:=FALSE;
  -- I/O package instantiation
  procedure PRINT_SCREEN;
  procedure DISPLAY;
  procedure HELP; -- permissible actions
  procedure PLACE_PROBE; -- in water
  procedure OPERATION;
end WQAU_P;
```

```
package body WQAU_P is
  procedure PRINT_SCREEN is separate;
  procedure DISPLAY is separate;
  procedure HELP is separate;
  procedure PLACE_ELECTRODE is separate;
  procedure OPERATION is separate;
end WQAU_P;
```

```
separate(WQAU_P)
procedure OPERATION is
  task MACHINE is
    -- as shown earlier
  end MACHINE;

  task body MACHINE is separate;
begin -- machine operation
  null;
end OPERATION;
```

Figure 4. WQAU package (partial)

Software Development

The software effort began as a student project in an undergraduate Ada course. The student was provided with a SOW and a photograph of the WQAU prototype. Development proceeded on a VAX 8800. Approximately 1200 lines of code were written to produce a working system level simulator. Through a summer student intern program, the software was rehosted on a government owned VAX-11/780 and then

to a PC/AT compatible machine using validated Ada compilers. Only one line of code was changed when moving to the PC due to a package scope anomaly. Once this transportability issue was resolved, the graphics package was modified to facilitate the use of color. The total software development effort required one (undergraduate) man-month.

EVALUATING THE SIMULATOR

The PC provided a mobile platform for demonstration and testing of the simulator. Except for the real analog probes (portrayed on the CRT), the graphics responded identically to the prototype hardware. The 15 second display of the five digit read-out, as specified in the hardware requirement, was consistently 15 seconds in the software simulation. This was accomplished through use of the Ada delay statement. Actual field tests of the hardware prototype revealed that soldiers were not happy with the duration of the delay. They were able to view and record a measurement in approximately 7 seconds and were ready to proceed to the next reading. The cost and development time to implement a hardware change would have to be weighed against a less than optimally human engineered unit.

In contrast, a change to the delay statement in the Ada simulator could be implemented on the spot by editing and separately compiling task body MACHINE. Changes at virtually no cost and in near real-time could be tested again for the human response to the reengineered condition. It is conjectured here that project managers would be prone to make SOW adjustments to improve system performance when changes early in the design phase would have little impact on budget and schedule.

Observations from the WQAU simulator in the laboratory have had a more profound effect than just the delay modification above. Software could guide the operator's steps to ensure that the CALIBRATE function was performed before MEASURE as stated in the operator's manual. This was not possible in hardware using a two position toggle switch. Thus a round of "what ifs" began to emerge as interaction with the simulator continued.

- what if a touch panel display replaced all mechanical switches?
- what if the software driving the display prohibited illegal courses of action?

- what if the simulator was downloaded to a PC laptop?
- what if an A/D converter interfaced the analog probe with the PC laptop?
- what if an off-the-shelf PC laptop costing \$1000 and weighing 12 pounds replaced a custom engineered unit costing \$5000 and weighing 45 pounds?

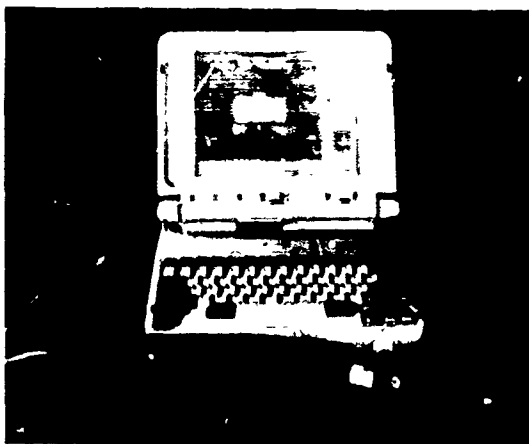


Figure 5. The WQAU Ada simulator hosted on a PC laptop.

The last hypothesis is shown in Figure 5. It does not take into account the A/D converter and weight of the probe, but estimates are good that this is a cost effective solution to the original SOW. The project manager might be expected to conduct a feasibility study for this course of action. In this project, the Ada PC/AT executable code for the simulator was downloaded without modification to a PC laptop computer (on which this paper was written) and used for inter-office demonstrations.

NEXT GENERATION SYSTEMS DEVELOPMENT

It is envisioned that future systems development will come to depend on software prototyping before committing scarce resources to build hardware prototypes. This is particularly true when there is a low probability of first pass success as system complexity and interoperability uncertainties increase. Further, families of common processor and display hardware may be developed which are reconfigurable. Systems will be characterized by coupling unique front ends to a mix of standard platforms under software control. Electronic warfare systems are likely initial candidates [Poza].

A reconfigurable set of common platforms is the cornerstone of the Army's future Armored Family of Vehicles and the heart of the DoD non-developmental item philosophy. Electronic systems are even more adaptable than mechanical systems to a "software prototyping before metal bending" methodology. With software reconfiguration and control, the simulator itself, carefully adjusted early in the design phase, has the potential to become the platform for the developed system.

Currently, computer-aided tools are being developed which will synthesize software descriptions to silicon compilers for the automated generation of application specific integrated circuits (ASICs). With these tools, software simulation may become a computer-aided project management gateway to a totally integrated design and fabrication process.

CONCLUSION

Ada provides a rich set of abstract data types which allows for high level modeling and simulation of future electronic systems. Because of the ability to write an interactive graphics package in Ada, functional behavior of systems can be represented in a visual display suitable for testing and analysis before the system hardware is built. System testing under software control provides a rapid prototyping feedback mechanism for the project manager to modify the statement of work early in the design phase before lengthy and costly developmental hardware prototypes are built. Such simulators can give insight to common hardware platforms which can be reconfigured under software control. These common platforms become system specific when adapted to unique front end sensors. The Ada programming language facilitates modeling, portability and reconfiguration when equipped with a graphics capability and when separate compilation is exploited.

REFERENCES

- Letter Requirement for the Water Quality Analysis Unit - Purification, U.S. Army Training and Doctrine Command, '3 Dec 85.
- Happel, Mark D. and Petrasko, Brian E., "Ada Tools for the Description and Simulation of Digital Signal Processing Systems". Proc. Sixth National Conference on Ada Technology, NTIS, 1988.
- Poza, Hugo B., "Ada: Maybe Not So Bad After All", Journal of Electronic Defense, Dec 88.



Kevin J. Cogan is a lieutenant colonel in the US Army Signal Corps. He received a B.S. degree from the US Military Academy and an M.S. degree in Electrical Engineering from Columbia University. He is graduate of the Army Command and General Staff College.

In addition to tactical command and staff assignments in the US and Europe, he was the Ada course director at West Point. A year of research at Duke University preceded his present assignment with the Electronics Technology and Devices Laboratory, US Army Laboratory Command.



Dr. Philip W. Caverly received his B.S. degree in Applied Mathematics and Engineering from Stevens Institute of Technology, M.S. in Applied Mathematics from Seton Hall University and his Ph.D. from New York University in Scientific Computing.

He is currently Chairman of the Department of Computer Science and Director of the Ada Technology Center at Jersey City State College. He is a consultant to government and industry in the areas of Ada technology, requirements engineering, CASE life-cycle tools, and methodologies for large system software development.



Mr. Carmine Marino will receive a B.S. degree in Computer Science and a B.S. degree in Mathematics from Jersey City State College in May 1989. He was the principal Ada programmer for this project. He was selected by the Army to serve as a software engineer at Ft. Monmouth during a summer intern program sponsored by the Southeastern Center for Electrical Engineering Education (SCEEE).

THE ADA SOFTWARE DEVELOPMENT METHODOLOGY EVALUATION
AND SELECTION PROCESS: FACT OR MYTH?

by

Sterling J. McCullough

Computer Technology Group, Ltd.
Washington, DC

ABSTRACT

One of the most important activities that is performed during the development of a software application is the selection of a software development method for use on the project. The choice of a method has a major impact on the quality of the resulting design, implementation, and documentation and on the productivity of the project personnel.

During a recently completed study, the author found that a number of the Ada developers that were interviewed had only a minimal understanding of the approach to be used in selecting a method for use on an Ada project. The purpose of this paper is to present a proposed approach for evaluating and selecting a software development method for use on an Ada project.

1. BACKGROUND

From November 1987 to May 1988, Computer Technology Group (CTG) provided support to Soncraft, Inc. in performing a methods study for the Center For Software Engineering, U.S. Army Communications and Electronics Command (CECOM). The study was entitled "Methodology Study For Real-Time Ada Problems" [Soni88]. The purpose of this study was to perform a theoretical analysis of the impact of a method on a set of generic Ada problems that had been identified by Soncraft in a previous study for U.S. Army CECOM [Soni87]. The theoretical results were then compared to actual results obtained from interviews with Ada developers that had current or recent experience in the development of real-time embedded Ada applications.

The study results which formed the basis for this paper were:

- 1) The method(s) selected for use on the projects had an impact on a large number of the generic Ada problems, particularly in the area of project management.
- 2) Many of the interviewees had a minimal understanding of the approach to be used in selecting a method to be used on an Ada project.
- 3) Most of the interviewees had selected methods for use on their Ada projects based primarily on subjective factors such as familiarity with the method, word of mouth, or recommendations from vendors.

2. STATEMENT OF PROBLEM

After analyzing the study results, CTG felt that the results pointed out a potentially widespread need for education and training in the evaluation and selection of methods for use on Ada projects.

3. OBJECTIVES

To address the problems stated in Section 2, the author then performed his own study titled "Evaluation And Selection Of Methods For Use On Ada Projects" [CTG88]. The purpose of this study was to develop a proposed approach for evaluating the suitability of a method for use on an Ada project.

The objectives of the Ada Methods Evaluation and Selection Study were to:

- 1) Develop a set of project characteristics to be used in categorizing the type of application to be developed and establishing the relative importance of the method features for the evaluation.
- 2) Develop criteria to be used in evaluating the suitability of a method for use on an Ada project, based on a set of method features.
- 3) Develop an approach for using the method features and project characteristics to evaluate the suitability of a method for use on a particular Ada project.
- 4) Provide a quantitative means to determine the suitability of a method for use on a particular Ada project. This information can be used to support or establish the validity of a method selection that was made based primarily on qualitative (subjective) data.

4. OVERALL APPROACH

The task of determining the suitability of a method for use on an Ada project is dependent on a number of factors such as the orientation of the method, the characteristics of the project, the experience and expertise of the project personnel, and the project management requirements.

It was important to the author that the approach to be developed for evaluating and selecting a method was comprehensive enough to be useful in a real-world environment and flexible enough to be used for a variety of project scenarios.

The proposed approach for method evaluation and selection is as follows:

Step 1. Develop a project profile which describes the features of the project that are important in the method selection process for the Ada project.

Step 2. Use the established project features to establish priorities and weights for the generic method features that will be used to evaluate the candidate methods. These method

features are now specific to the particular Ada project.

Step 3. Set up a checklist for use in evaluating candidate methods. The checklist contains the prioritized and weighted method features and provides a quantitative measure of the importance of each feature for the particular Ada project.

Step 4. Establish the criteria to be used in selecting one of the candidate methods based on the results of the method evaluation.

Step 5. Select the candidate methods that are proposed for use on the Ada project.

Step 6. Evaluate the candidate methods based on the prioritized and weighted method evaluation features.

The result of the method evaluation process is that each candidate method will receive a quantitative rating (score) which reflects its suitability for use on the Ada project under consideration. This information is then used, usually in conjunction with other information, to select the method to be used.

5. PROJECT PROFILE

The purpose of the project profile is to establish the features of the Ada project that will have a significant impact on the selection of a method. These features are grouped into the following areas:

- * Application characteristics
- * System goals/attributes
- * Project characteristics
- * Personnel characteristics

A sample set of these project features is supplied in Figure 1. The sample set can be used as a base from which to develop the set of features that reflects the specific needs and concerns of the project under consideration.

The Ada developer then decides which features are relevant for the project and rates the project features to reflect the actual project environment. An actual project profile is provided in Figure 2.

6. METHOD FEATURES

Once the project profile information has been specified, the next step is to use this profile information to establish the method features which are important for the Ada project.

Figure 3 contains a proposed set of generic method features that can be used as a basis for evaluating a candidate method for use on an Ada project. This list of features covers a range of issues to include methodology implementation details, project management, and automated method tools. A developer can add issues to the list, as required, to provide a more comprehensive set of method evaluation criteria.

The generic method features are then ranked to establish the relative importance of the method features for the Ada project under consideration. The project profile information is used to ensure that the importance of the method features reflects the information obtained from the project profile.

Figure 4 contains an actual ranking of the importance of selected method features for an Ada project. The rankings for the method features reflect the actual project profile information presented in Figure 2. This list or one similar to it is then used to develop a checklist for evaluating candidate methodologies.

7. METHOD EVALUATION

The ranking of method features (Figure 4) is used to prepare a checklist for use in evaluating the methods that are candidates for use on the Ada project. Figure 5 contains an example of an actual checklist that was developed to evaluate candidate methods.

The result of each evaluation is a filled-in checklist which provides an individual rating of the candidate method for each method feature. Figure 6 contains an example of an actual filled-in checklist for an Ada project.

8. METHOD SELECTION CRITERIA

The result of the entire evaluation process is a quantitative rating

(score) for each method that is evaluated. The relative scores for the different methods can be used as input for determining which of the candidate methods is most suitable for use on the Ada project.

The selection of a candidate method can be made according to selection criteria that are set up by the Ada developer. For example, a developer could decide to throw out all methods which receive a score below passing (such as 70%). The remaining methods could be evaluated based on their overall score and their score on a selected subset of criteria.

9. SUMMARY

The evaluation and selection of a methodology for use on an Ada project is one of the most critical decisions that is made during the development effort. The approach that was presented here is just one possible approach that can be used perform this activity.

This approach provides a set of quantitative and objective results that can be used in conjunction with qualitative and subjective results to evaluate the suitability of a method for use on an Ada project.

It should be noted that the success of this method depends heavily on the accuracy and completeness of the project profile information and the method features.

REFERENCES

[CTG88] S. J. McCullough, "Evaluation And Selection Of Methods For Use On Ada Projects", 1988.

[Soni88] S. J. McCullough, F. Franci, C. Johnson, "Methodology Study For Real-Time Ada Problems", Final Technical Report To US Army Research Office, 27 June 1988.

**FIGURE 1: SAMPLE PROJECT PROFILE
FEATURES**

APPLICATION CHARACTERISTICS

Real-Time
Distributed
Concurrency
Mission-Critical
Memory-Critical
Time-Critical
Secure System
Embedded Application
Data-Oriented
Process-Oriented

POSSIBLE RATINGS

Yes or No
Yes or No
Yes or No
Yes or No
Yes or No
Yes or No
Yes or No
Yes or No
Yes or No
Yes or No

SYSTEM ATTRIBUTES

Reliability
Maintainability
Traceability
Efficiency
Portability
Reusability
Flexibility
Verifiability
Expandability
Accuracy
Integrity
Modularity

POSSIBLE RATINGS

Low, Medium, High
Low, Medium, High
Low, Medium, High
Low, Medium, High
Low, Medium, High
Low, Medium, High
Low, Medium, High
Low, Medium, High
Low, Medium, High
Low, Medium, High
Low, Medium, High
Low, Medium, High

PROJECT CHARACTERISTICS

Size - LOC
Size - Personnel
Complexity
Risk
Development Time
Development Cost
Project Visibility
Use Of Automated Tools
Training Budget
Customer Influence

POSSIBLE RATINGS

Small, Medium, Large
Small, Medium, Large
Low, Medium, High
Low, Medium, High
Short, Medium, Long
Small, Medium, Large
Low, Medium, High
Low, Medium, High
Small, Medium, Large
Low, Medium, High

PERSONNEL CHARACTERISTICS

Work Experience
Related Work Experience

POSSIBLE RATINGS

Low, Medium, High
Low, Medium, High

FIGURE 2: ACTUAL PROJECT PROFILE

APPLICATION CHARACTERISTICS

Real-Time
Distributed
Concurrency
Mission-Critical
Memory-Critical
Time-Critical

ACTUAL RATINGS

Yes
Yes
Yes
Yes
Yes
Yes

Secure System
 Embedded Application
 Data-Oriented
 Process-Oriented

Yes
 Yes
 No
 Yes

SYSTEM ATTRIBUTES

Reliability
 Maintainability
 Traceability
 Efficiency
 Portability
 Reusability
 Flexibility
 Verifiability
 Expandability
 Accuracy
 Integrity
 Modularity

ACTUAL RATINGS

High
 High
 Medium
 High
 Low
 Low
 Medium
 High
 Medium
 High
 High
 Medium

PROJECT CHARACTERISTICS

Size - LOC
 Size - Personnel
 Complexity
 Risk
 Development Time
 Development Cost
 Project Visibility
 Use Of Automated Tools
 Training Budget
 Customer Influence

ACTUAL RATINGS

Medium
 Medium
 High
 High
 Medium
 Medium
 High
 Low
 Small
 Medium

PERSONNEL CHARACTERISTICS

Work Experience
 Related Work Experience

ACTUAL RATINGS

Low
 Low

FIGURE 3: GENERIC METHOD FEATURES

TECHNICAL

Process Visibility
 Data Visibility
 Ada-Oriented
 Information Hiding
 Program Structure
 Data Structure
 Quality Of Resulting Design
 Design Consistency
 Problem Definition
 Use For Large Projects

IMPORTANCE

Low, Medium, High
 Low, Medium, High
 Low, Medium, High
 Low, Medium, High
 Low, Medium, High
 Low, Medium, High
 Low, Medium, High
 Low, Medium, High
 Low, Medium, High
 Low, Medium, High
 Low, Medium, High

MANAGEMENT

Maturity Of Method
 Ease Of Use
 Ease Of Learning
 Documentation

IMPORTANCE

Low, Medium, High
 Low, Medium, High
 Low, Medium, High
 Low, Medium, High

Life Cycle Model Flexibility
Available Training
Guidelines For Method Usage

Low, Medium, High
Low, Medium, High
Low, Medium, High

AUTOMATED TOOLS

Availability Of Tools
Cost Of Tools

IMPORTANCE

Low, Medium, High
Low, Medium, High

FIGURE 4: ACTUAL RANKED METHOD FEATURES

TECHNICAL

Process Visibility
Data Visibility
Ada-Oriented
Information Hiding
Program Structure
Data Structure
Quality Of Resulting Design
Design Consistency
Problem Definition
Use For Large Projects

IMPORTANCE

High
High
High
High
High
Medium
High
Medium
Medium
Medium

MANAGEMENT

Maturity Of Method
Ease Of Use
Ease Of Learning
Documentation
Life Cycle Model Flexibility
Available Training
Guidelines For Method Usage

IMPORTANCE

High
High
High
Medium
Low
Medium
High

AUTOMATED METHOD TOOLS

Availability Of Tools
Cost Of Tools

IMPORTANCE

Low
Low

FIGURE 5: SAMPLE METHOD EVALUATION CHECKLIST

METHOD FEATURES

PRIORITY

WEIGHT

Ada-Oriented
Ease Of Use
Ease Of Learning
Available Training
Guidelines For Method Usage
Quality Of Resulting Design
Process Visibility
Information Hiding
Program Structure
Maturity Of Method
Data Structure
Documentation

1
2
3
4
5
6
7
8
9
10
11
12

20%
15%
15%
10%
10%
7%
7%
5%
4%
3%
2%
2%

100%

**FIGURE 6: ACTUAL METHOD EVALUATION
CHECKLIST**

<u>METHOD FEATURES</u>	<u>PRIORITY</u>	<u>WEIGHT</u>	<u>SCORE</u>
Ada-Oriented	1	20%	10%
Ease Of Use	2	15%	10%
Ease Of Learning	3	15%	7%
Available Training	4	10%	5%
Guidelines For Method Usage	5	10%	7%
Quality Of Resulting Design	6	7%	3%
Process Visibility	7	7%	4%
Information Hiding	8	5%	4%
Program Structure	9	4%	3%
Maturity Of Method	10	3%	3%
Data Structure	11	2%	2%
Documentation	12	2%	1%
		-----	-----
		100%	59%

BIOGRAPHY

Sterling J. McCullough is President of Computer Technology Group, Ltd and has over ten years of experience in the development of software for both Government and commercial applications.

Sterling has over seven years of Ada development and analysis experience and has spent the last two years performing studies and research in the evaluation of methods for use on Ada projects.

Sterling received a BS degree from the University Of Notre Dame and an MS degree for Carnegie-Mellon University.



AN ADA DESIGNED DISTRIBUTED OPERATING SYSTEM

Martin B. Serkin

Martin B. Serkin & Company

Abstract

The design of a three level distributed executive for an avionics system using Ada was a difficult undertaking. It required a new understanding in the approach to the design of a system as well as training not only in the language but in the entire approach to the problem. Object oriented design was to be used and required a new approach to the problem. Personnel were not trained in the language nor was management prepared for such an undertaking. It required perseverance as well as a commitment on management and personnel to get rid of the old approaches and try the new.

Introduction

In the last quarter of 1994 the United States Air Force requested from the major air frame manufacturers a design called the Three Level Executive, these levels being a Kernel Executive, a Distributed Executive and a Systems Executive. This operating system was to be designed Generically. For this paper a generic system is as one that, by use of data types, the configuration of hardware, communications and application tasks will define to the Kernel and Distributed Executive packages their operating environment. By changing these types and then re-compiling these packages the executive is ready for operation. There is one modification to this operation. It is those portions of the system that are hardware dependent and must change when the host computer changes. This will be discussed further in this paper. The operating system is to be input table driven and these tables would describe the hardware configuration, the communications network, the necessary applications, and any other pertinent information necessary to make a functional operating system for aircraft now and in the future.

A brief explanation of this avionic distributed operating system is necessary in order to familiarize the reader with the terms in this paper. The system is designed to run cyclically. By cyclically we mean that there is a set major cycle, i.e., 64 hertz and the major cycle is broken down into 64 minor cycles each of length 15.625 milliseconds. Certain tasks will be scheduled to run during each minor cycle, every 15.625 milliseconds such as the Distributed Executive while other tasks can run each cycle, every other cycle, once a major cycle, etc. The Kernel and Distributed Executive are in each processor in the system while the Systems Executive and associated other tasks, such as the Configuration Manager will be in just two processors.

Basic Design

The design is rather complex in that the Executive can control up to N processors, up to Y different busses, with the types of busses being independent of each other, recognize bus errors and then determine faults, and do dynamic relocation of those tasks that are affected by the faults while keeping the rest of the system operational with a minimum of interference to the general operation of the total system. The following is a brief description of each of the three parts of the Three Level Distributed operating system.

Kernel Executive

Ada compilers come with a runtime kernel and runtime system routines. The supplied kernel was not applicable to our design and required extensive rewriting. This included a design of a new linker for reasons explained further in this section. The Kernel is responsible for the following functions:

1. Initial power-up testing of the processor.
2. Initialization of the interrupt services the name used to define the IO handler routines.
3. Calling the Distributed Executive to do the bus initialization.
4. Creating the task control blocks as tasks are loaded into their respective processors.
5. Handling Ada exceptions.
6. Scheduling of Tasks on a cyclic basis.
7. Maintenance of timers, clocks, page registers, etc.
8. Participating in the reconfiguration.

Distributed Executive

This is the name used to define the IO bus handler. The Distributed Executive, hereafter referred to as the DE, has the following responsibilities:

- A. During System Initialization
 1. Build the system network by seeing what processors and devices are connected to each bus.
 2. Load the processor connected to the System Mass Memory, the System and Configuration Manager applications.
 3. Read the configuration tables, application load instructions, active remote terminal tables and all other system tables off the System Mass Memory.
 4. Send these tables to all other processors in the system.
- B. During System Running
 1. As a bus controller, send out commands to do the required input/output for each cycle.
 2. As a remote processor, prepare the bus to receive or send data for the current cycle.
 3. Process System Mass Memory requests.
- C. During Re-configuration
 1. Do the System Mass Memory IO to load the processors with the required application tasks.
 2. Insure that the processors not involved in the re-configuration continue their cyclic IO.
 3. Communicate to the Kernel when tasks are deleted and when a load has been completed.

System Executive

The System Executive is a relocatable task that works with the Configuration Manager and the Maintenance Monitor to provide the following functions:

1. Handles all error reports and passes them to the Maintenance Monitor. The Maintenance Monitor will determine when errors become faults. The Maintenance Monitor will not be discussed in detail as it is system related and will change with each major application that uses this executive.
2. Control access to the System Mass Memory on a task priority basis.
3. Gathers system status from all the processors and prepares data which is passed to the Maintenance Monitor to determine processor task/bus failure.

This brief explanation should suffice to give the user a working knowledge of the way the system is designed. The next section will go into a bit more detail on the design of the two executives that are the most sensitive to the multi-processor environment.

Distributed Executive Sensitivity

The DE, being the controller of all the busses in the system, is the most sensitive to the multi-processor environment. When the Kernel Executive, in each of the processors, finishes its initialization, it calls the DE to determine the hardware configuration of the system using tables that are on the System Mass Memory, hereafter called the SMM. These tables are a series of arrays that contain the complete description of all hardware that is on each bus, the bus configuration and bus types, IO channels and the initial application load of each of the processors.

Upon receiving control from the Kernel Executive, the DE will read the backplane id of the processor. The backplane is compared to the array that contains the backplane to processor identification. This array also specifies whether this processor is to be the initial controller of the SMM and therefore the processor that will contain the primary System Executive. If this is the Primary Processor, the DE will now acquire, from the SMM, the following arrays:

- A. The hardware configuration of the system. The DE will use this array to issue bus tests to determine which of the remote processors/hardware is responding to the commands being issued on the bus. A system status table is constructed to hold the results of these tests.

B. The array containing the processor to remote terminal identification is acquired. This array contains the remote identifier for each of the remote terminals to the bus controller.

C. The message to task array is acquired. This array contains the relationship between message identification and remote terminal identifiers. This array is used to pass data between remote terminals. Since this is a distributed system an application does not know where another application resides. It is the responsibility of the DE to command the designated bus to send or receive the data required. The information about what application has commanded a read/write of a particular piece of data is located in this array. Since hardware, i.e., sensors, radar, etc., can not issue these requests, this array contains their identifiers and information about when this data is to be written, the remote id of the hardware and other pertinent information.

When the arrays are read off the SMM into the primary processor, the DE will send all these arrays to all other processors that can become the bus controller. The System Executive and Configuration Manager is loaded into the primary processor. At this point a message indicating that the System Executive has been loaded is sent to all other processors in the system. At this point control is returned to the Kernel Executive.

During primary processor initialization the other processors are waiting for the load of the configuration arrays and the message that indicates that the System Executive has been loaded. If this message is not received, another processor will become the primary processor, based on a series of parameters and time calculations, and perform the services described above. Once the configuration tables have been received, the other processors will be waiting for application load messages and finally the Initialization Complete message. Once the Initialization Complete message has been received, the entire system is put in a system running state.

Each application, on initialization, will call the DE via a Kernel call and will register its cyclic Input/Output requests. This request will contain all necessary information about the data to be sent or received. It will also contain the cycle period when the data is to be sent/received, the

number of words in the Input/Output area and other necessary information. These requests are processed by the DE and the cyclic IO tables are built for each of the cycles in the system.

The Input/Output task of the DE is scheduled every 64 hertz. It will do all necessary processing to create the Input/Output commands for this cycle. Once the IO chain is built, the DE will issue the commands to the physical bus hardware to start the IO chain. When errors are detected by the Interrupt Service routine of the DE, these are reported to the Maintenance Monitor. The DE will not re-direct Input/Output or declare any device inoperative. This is done only under direction of the Configuration Monitor. Each cycle, the DE will interrogate each processor to see if any application has requested any services. This is done by each of the bus masters and will differ with the type of bus, 1553B, High Speed data busses, etc.

System Executive Sensitivity

There are two System Executives always live in the system, the Primary which will receive data and write data and the stand-by which just receives data. If the processor containing the Primary System Executive should fail, the stand-by will assume control and will send a message to the Configuration Monitor that a relocation should take place. Since the System Executive controls access to the SMM, it is important that there always be an active System Executive and along with it the Configuration Manager.

The design of the System Executive and Configuration Manager is open-ended since these tasks are more dependent of the type of system, fighter, transport, space flight, etc., than the Kernel and DE who are dependent of the hardware.

Design Configuration

In order to design the system, an ideal configuration of hardware was designated. This configuration consisted of the following equipment:

1. Eight 1750s designated as Mission Data Processors, each containing one million words of storage.
2. Four 1750s designated as Vehicle Data Processors.
3. A Common Signal Processor to collect the data from the sensors which were connected to 1553B busses and send

this data over the designated Mission and Vehicle busses.

4. Three dual redundant high speed data busses; one pair allocated for the SHM busses; one pair allocated as the Mission Data Processor communication path and the last pair for mission to vehicle data processing communications.

5. A quad-redundant set of high speed data busses for communications between the Vehicle Data processors which were used for voting purposes.

Our design was based on this configuration. When implementation of the system was started this configuration did not exist. We were forced to redesign the system to a somewhat scaled down configuration which consisted of the following:

1. Two 1750 mission data processors with 64k of storage, connected via a 1553 data bus.

2. A VAX 780 simulating sensor data communicating over a 1553 bus.

3. A Harris computer simulating the cockpit displays, also over a 1553 bus.

This configuration caused much rethinking in the DE and made our original design somewhat impossible to carry out. We had no off-line processor to build the configuration tables and this had to be simulated and because of memory requirements much of the generic executive had to be eliminated.

ADA Implementation

Now that a brief and I hope not too boring description of the Three Level Executive has been presented, we will discuss the use of Ada and its benefits and problems, which I am sure most of those reading this paper, and I hope there are many, want to know how Ada was applied to this subject. Ada was chosen as the language of implementation since the Department of Defense has decreed this is the High Order Language of Choice. This was decreed by the Advanced Avionics System which was the original contract that was let to design this executive. The other reason for choosing Ada is described by Mr. Grady Booch in his book "Software Engineering with Ada" and let me quote:

"Ada is more than just another programming language, however. Along with the Ada Programming Support Environment, it represents a very powerful tool to help us understand problems and

express their solutions in a manner that directly reflects the multidimensional real world."

Since Ada and Ada compilers for military computers, mainly the 1750A, which was our target computer, were not many in number nor was there many available trained personnel, we first had to come up with an approach which would make use of the best parts of Ada, direct all design to be object oriented and avoid, at all possible cost, ADATRAN design. By ADATRAN is meant the design of a global database, all applications sharing common data, tasks communicating among themselves without going through the executives, and all other bad techniques that we as an industry have used over the years. In other words we did not want to make the mistakes of old. The system had to be easy to maintain, and be as modular as possible.

Education of Management and Staff

The following is an outline of our approach to training management as well as the current staff in using Ada and implementing the design method called Object Oriented Design.

In the great and old days, systems were designed where applications had free reign to pass data directly from one application to another. Programs were designed to fit in a single computer and there was no need for a different approach. Just store it in a common area and everyone and his brother could get a look at it and possibly destroy, or modify the data. This executive had to insure data reliability and integrity. This hurdle was one of the most difficult to overcome in our retraining of management and staff. This is a distributed system and we had to install into the applications and system engineering staffs that unless there was a shown need, no applications could be guaranteed to reside in the same processor.

Again to digress, before Ada, systems engineering was done for a project with little or no regard to the language to be used because it did not affect the operations of design, coding, programming or documentation. If some of it was in JOVIAL, some in Fortran, a little in C, a bit more in assembler, so what. This entire approach to the design of a project had to be changed. The software system had to be designed with the capabilities of Ada and its constraints and viewed in its entirety

with each object, radar application, Nav, etc., being designed as a self-contained package with the data messages being the means of communication between other applications.

We now had to take a software engineering approach to the problem and think a bit differently. Each object, that is for example a Navigation system, a weapons system, a sensor system, etc., was to be designed as packages. These packages did not have to take into account where other applications resided. Each package was to be designed without regard to the other packages nor worry about the common areas for data passage. There was to be little or no shared data. All information is to be passed via the DE and the DE has the responsibility to get the information to the correct application data area. Obviously input and output interfaces still existed but data was not to be directly placed in any common area except for system related data. See, there is always an exception to the rule. All parameters had to be typed and all limits to arrays, ranges and any other types that were greater than 1 in length were to be parameterized.

It sounds simple but people just want to do it the old tried and true way. The usual argument we meet was "hey, I know the array is 1 to 10 items so why not define it that way. It's easier to read than some silly object 'first to object' last." Maybe easier to read but harder to maintain, change and if more than one procedure uses the array how do we insure the same ranges. People consistently tried to avoid using the Ada language to its full extent.

The toughest problem was the instilling in the group that Ada was to be viewed in a Software Systems Approach. This meant that from the very beginning of a project, all design starting with the system requirements, must be viewed in the totality of the Ada language. All documentation had to be prepared with the constraints as applied to Ada. As is standard, in the government's wisdom, the documentation had to be done to Military Standard 2167 which is not too well defined for the Ada language. Second, most of the personnel had never seen Ada, let alone the rigid typing and packaging that we were trying to get implemented. The system requirements had to be done before detailed design could begin so we used Mil-Standard

490 which permitted us more leeway in designing the documents.

All the personnel working on the project except for a few had never heard of, let alone worked in, any environment where one had to view each object, Radar, sensors, Kernel, DE, as a package with data that can be viewed by only those who must have access. This packaging of related procedures for a specific object was loosely used in other projects but had to be enforced by other means than the language itself. For an Ada system this was tightly controlled with the use of visibility. No application could "with" a Spec unless that Spec was part of its own package. A portion of the Kernel which is used by all applications to communicate with the operating system is also "withed". This is being eliminated with the implementation of a DEX call which is explained later in this paper. Now that we had our design specified, our system engineering specification clarified, our coding standards specified, we were ready to go. How long could it take to learn Ada and get this executive on the road? The staff was still untrained and what training problems could we encounter? We were all professionals.

Learning Ada

Our environment consisted of our target computers, Mil-Standard machines 1750As and there were three systems just waiting in the lab to be used. The VAX1 system was our host computer and it contained an Ada compiler. Management figured that Ada was like any other language and allowed forty hours of training to get those who were unfamiliar with the language up to speed. This was a gross underestimate of time. I would say that based on our experience with Ada, one should allow about 400 hours of training with active lab practice to have a person learn all the intricacies of the language. The group being trained had an average of over 10 years of experience with many languages so use this as a guide.

Since we could not convince our management that the time allowed was too small we just pushed ahead and designed, coded and tested out the executive as we learned. If there was a space and I could list the programs developed on our first attempt one would notice that we had produced Adatran in our code, all the design

and system engineering we painstaking laid out was violated and the system ran but not as generically as we wished. Most of this was attributable to the time constraint placed upon us to demonstrate a cycling system for upper management. Most of the code was HARDCODED to fulfill the requirements of the demo.

The breaking of bad habits was probably our toughest hurdle to get over. That is besides trying to learn the language. The Mil-Standard 815 Ada description was so clear that almost anybody could just read the text and become an Ada expert. It just so happened that our staff probably had learning disabilities and found the text somewhat difficult to grasp on a first reading. The entire staff went out and bought a book written by Mr. Grady Booch, "Software Engineering with Ada" 2 and used this text along with the Ada standard and this improved the learning curve tremendously.

The second problem to overcome was the idea of a distributed system. Most of the staff had not had experience with a distributed system. They wanted to use the global database, or compool structure and have tasks communicate directly with each other through shared memory areas. They also were not used to strict typing of items and tried to avoid this at all cost. A special problem existed for the Kernel and DE which was the use of Chapter 13 and all it implies. Since this was compiler manufacturer dependent, VAX Ada and our Ada for the 1750s were not too compatible. This caused and is still causing a problem in our development. More on this later.

The system was developed with a communication method, implemented in the DE, that allowed the application tasks to request the sending and receiving of messages. The DE would take care of placing the data into the user specified data area or send the data from the specified matter to or from the specified matter to or from the appropriate processor. Easy to lay down the rules, hard to get people to follow. A module was developed into which all our data types and message formats were described. All of a sudden, this module was growing with objects that were visible to the world. A quick stop was put to this and was placed under executive control to limit the amount of visible objects to the entire environment. We had to make the Kernel Executive have several specs consisting of the specifications

available for the application tasks, a specification for the DE and the Kernel's private specification. This came about because of problems with the private pragma not working and some other compiler problems.

After a period of time we considered ourselves an Ada trained Ada expert group. But where was our compiler for the target machines?

Ada Selection and Ada Tools

To refresh the reader's memory, this project started in the latter part of 1985 and there were not many Ada compilers available, let alone those that were certified by the Department of Defense and had the 1750 as a target computer. A group of analysts was selected to go out and test those compilers. They were compared on the following criteria:

1. DoD Certified.
2. Compile time based on a specified set of programs selected from the Dais mix.
3. Optimization capabilities.
4. Available support from the supplier.
5. Release of the source code for the Kernel and run-time support packages.
6. Cost and maintenance of the compiler.
7. Must be able to run on the VAX and produce 1750A object code.
8. Manufacturer supplied linker with the capabilities that were required for our executive.
9. The compiler with the least known bugs at the time of selection.
10. Debugging tools available along with any simulator that would run on the VAX and simulate the 1750 computers.
11. Chapter 13 implementation. What extras were available and how were they implemented. The executives had to use much of the Chapter 13 options and even though these are kept to a minimum, the implementation of these options were of great importance to the selection of the compiler.

The process went on for four months and we finally selected our compiler. I do not wish to specify the name of the corporation or those compilers that were selected since the testing was done over three years ago and I am sure that the ones not selected have improved and I feel that it would be unfair to the others now.

When we received our compiler, one of the staff had to take the Kernel and the supplied linker and make our changes to the code. This task was not simple and took slightly longer than expected. The changes were made over a period of time with work still being done.

As we started to carry out our design, one found that much of the features that make Ada a language which embodies much of the modern software development principles but also rigidly enforces them just was not yet implemented into the compiler. In our first delivered compiler representation clauses did not work. This would have been somewhat acceptable for most applications, but all the IO commands that had to be generated required bit manipulation and without it masses of memory was used. Packing also was not yet implemented and strings were a pain and were avoided if at all possible. The next non-implemented feature was Machine Interface, again a problem not only for the DE but for the Kernel Executive. We had decided from the beginning to attempt to use little or no 1750 assembly packages. In the beginning not possible. At present a new delivery of the compiler contains these features and we are implementing the changes as I write this paper.

Of course we found compiler bugs, which is expected as one tries to use as many of the features of Ada and its constructs as possible. Why you ask, because we were attempting to become Ada experts and to do this you try everything in the book, wouldn't you?

Our next implementation problem was the linker. Since we had to be able to load many application tasks and each task loaded had to be memory protected from the other tasks, the linker had to be changed to implement the page register scheme available in the 1750s. At present our linker will separate operand from instruction to take advantage of the page register system in the 1750s, and we are now trying to install the BEX linkage for applications to communicate to the executives instead of a call.

For those unfamiliar with the 1750 architecture, the machine has a series of page registers that permit the operand and instructions to be separated and protected on a page boundary. A page boundary consists of 4096 16 bit words. Each application is permitted to have as much as 64k of

operand that it can access and a similar amount of instruction space. The BEX linkage is an executive call which causes an interrupt which is processed by the Kernel. With the implementation of this to the Kernel Executive, those pages containing the executive will no longer be visible to the user. This would permit the application to have the full use of the paged memory available.

The simulator received with the compiler did not quite meet our needs. A 1750 simulator was developed to run on the VAX host using the output of the modified linker. This worked well and still does for the testing of applications tasks but for the Kernel and DE very limited testing could be done. We were forced to do our testing on the 1750s.

As expected there were very little tools available for the testing of Ada generated code for the 1750 computers. A simple debugger with very limited capabilities existed and the debugging time greatly exceeded our time estimates. When we finally were able to test, one found the usual number of compiler bugs which then caused the programmers to start blaming every bug on the compiler. Is not human nature so predictable? Give a programmer a straw and they will make it into a giant oak. As a generic system that is being funded from IR & D funds our budgets were not high nor our equipment the best. The programmers did have a little bit to complain about.

Several demonstrations were given to management and lo and behold a real three level executive was being developed and tested on a limited number of computers using several busses. The idea of a DE written in Ada and running on Mil-Standard equipment was fully available now for practical use.

Implementation of the Final Three Level Executive

The corporation was now convinced that the project could be done. No more demo systems had to be developed to prove that the design and specifications could be met. The old 1750s disappeared and new equipment with real debugging consoles, appeared. Funding was approved to make this executive a truly generic system. The old adage comes true again, there never is enough time or money to do it right the first time but there is always time and money to do it over.

Currently the staff is producing a truly generic system. It is table driven so that the number of processors, busses and bus types, and reconfigurability is regulated and can be changed by simply changing the parameters, recompiling and off we go with maybe a little testing. The Three Level Executive is now running on a three processor system. It is being modified for us a government contract involving a processors, four busses and a large number of sensors. The busses being used are Mil-Standard 1553B.

To add different types of busses, the system allows the insertion of that bus handler with a minimum of changes. The DE checks the bus type that is to be used and will call the appropriate bus package based on that type. As bus types are added, the bus package will have to be debugged but this does not affect any other packages in the system.

Projectizing the Executive

In the latter part of 1987, the organization that I am contracting with was rewarded a contract to supply the software for a retro-fit of an existing airplane for the Air Force. Our executive has been chosen to be used in this project and is also being proposed for a variety of systems being implemented or planned for in the future. The immediate project should be discussed, though somewhat briefly as this paper is getting too long. Upper management, in its great wisdom, decreed that the Three Level Executive be used in a project just won by the company. Without consulting any of the Executive group, the work began. The people on the project were not Ada trained, had no knowledge of object oriented design and thought a package was something you brought home from the store. The systems and the operational flight software personnel had no knowledge of our system but prepared the design of all the software without any working knowledge of the operation executive.

As you can guess, they designed a complete ADATran system, with naming conventions, shared databases, disregard for relocation and all other things this executive expected to be used. As one can surmise, all the problems that the executive group went through during design and implementation is now going on once more.

The new project was designed without regard for the Ada language and therefore is having difficulty in adapting to our executive. The Air Force required document specification Mil Standard 2167 does not in any shape or form allow you to document Ada packages as one should. It is still designed to JOVIAL with all its faults and difficulty to read. For example, we had to re-write our detailed design to conform to 2167 and the document was over 2000 pages in length. This might be all right for a lifting exercise but as a reference document it is almost useless.

The Executive Group is now working with the project to try to retrofit our executive into their design. The best analogy I can think of is when a large car manufacturer saw a need for diesel engines for their cars, instead of designing a new engine they just took their gas and made it into a diesel. I am sure we all remember the results of that fiasco. As of the writing of this paper, we are still battling the hard-liners who want global databases, event-driven IO and all the Adatran that you could possibly implement. The configuration for the project consists of the following:

1. Five 1750 computers connected via 4 1553 busses.
2. Sensors connected to the 1750s via 1553 busses.
3. One of the 1750s will use memory as the SMM.

The training goes on as well as the instilling of Object Oriented design in all personnel working on the project. Management is coming around to the idea of software engineering and control and the strength of the Ada language in controlling a project.



Martin B. Serkin
Martin B. Serkin & Company
16479 Halsey Street
Granada Hills, CA 91344

Mr. Serkin received his Bachelors degree in Psychology with a minor in mathematics from Brooklyn College in 1965 and received his MDA from Monmouth College in 1978. He has been involved in the real-time operating system field for over 28 years and has been a consultant for the past 8 years working in the aerospace field as well as doing a real-time system for NATO in Europe.

PARSIM: A Parallel and Real-Time Simulator for Concurrent Programs

Don M. Coleman

Ronald J. Leach

Department of Systems and Computer Science
School of Engineering
Howard University
Washington, D.C. 20059

1. Abstract

This paper describes a Parallel and Real-Time Simulator (PARSIM) which was developed in Ada using the Meridian compiler on an AT&T PC 6310. PARSIM, developed as a software tool, uses a Petri net model for simulation of concurrent computational algorithms. The paper discusses the structure and operational characteristics of PARSIM.

2. Introduction

The study and analysis of real-time and parallel systems is an active and important area of computer science. An important tool for such studies is the Petri net [Peterson, Cherry]. A Petri net is a particular type of graphical model which is especially suited for the representation of concurrent and parallel systems. In this paper we report on a Petri net based Parallel and Real-Time Simulator (PARSIM) which was developed using the Meridian Ada compiler on an AT&T PC 6310, which is based on the INTEL 20286 microprocessor and which uses MS-DOS. The paper consists of 3 major parts. First we briefly describe Petri nets, the tool for modeling systems; next we outline PARSIM, its components, parameters, and the rules for execution; finally, we discuss results on the execution effectiveness of PARSIM.

3. Description of the Petri net model

A Petri net is a special type of directed graph consisting of nodes, transitions, and arcs. Symbolically nodes (also called places) are represented by circles, transitions by vertical bars, and the flow in the graph by directed arcs. In addition, Petri nets use tokens, indicated by dots in the nodes, to control the execution of the network.

Execution of a Petri net may be described by indicating the state of the network as a function of time. A convenient way of defining the state of a given net at a particular time is to describe the number of tokens in each place at that time. Changes in state are controlled by the distribution of tokens in the net. State changes occur as a result of firings of transitions. When a transition fires, there is a redistribution of the tokens associated with its input and output places. A node with an arc directed into a transition is defined as input place for that transition; whereas an output place is defined as one with an arc directed from the transition to that place. A transition can fire (be enabled) when each of its input places has at least one token for each arc directed from that place into the transition. A transition fires by redistributing the tokens among the places as follows:

One token is removed from an input place for each arc between that input place and the transition;

One token is created and deposited into each output place for each arc from the transition into the place.

Figure 1a and 1b illustrate the result of the firing of transition t_2 . For this particular net we have four transitions and six places. The initial state is given by:

$(1, 1, 0, 0, 0, 0)$

where the i^{th} entry in the six-tuple represents the number of tokens in the i^{th} place. After transition t_2 is fired the new state vector is given by

$(0, 2, 0, 0, 1, 0)$.

The time associated with the firing of a transition is assumed to be zero within the class of "Untimed Petri Nets" (see [Peterson]). Also for this class of nets, the sequence of firings of the transitions is random; i.e., a given transition may fire whenever it becomes enabled; however, it is further assumed that the probability of two transitions firing at the same time is zero. A useful

characterization for a Petri net and a given initial state vector is the reachability set. This set defines all the states which can result from any sequence of transition firings starting from the initial state. Algorithms exist for the construction of the reachability tree, which is a graphical representation of the reachability set (see (Peterson, Holliday & Vernon)).

4. The Structure of PARSIM

The Petri net is a natural tool for the modeling of concurrent systems. The correspondence between the simulation model and Petri nets is as follows. PARSIM is a discrete event simulator. That is, it is based on a model of discrete processes. For our purposes, a process is an activity which executes or proceeds over time having a well defined beginning and end. Since each of the transitions represents a process in the simulation, we introduce the idea of time associated with the transition firing duration. The initiation and conclusion of processes (transition firings) define events in this model. Events change the state; i.e., the distribution of the tokens in the net. Recall that places and transitions are Petri net primitives. We will be concerned with a particular interpretation of the net which requires us to expand the definition of state to include information about the "execution" of transitions. Primitive transitions do not execute; they only fire instantaneously. We can represent this new notion, "execution of a transition", in terms of primitives as shown in Figures 2a and 2b.

We see in Figure 2a a representation of transition T_1 with its input and output places P_1 and P_2 . When we interpret this transition as executing, we mean that the structure of the transition is as shown in Figure 2b where transition T_1 is expanded into two "virtual transitions" and a "virtual place" represented by T'_1 , T''_1 , P' respectively. The firing of T'_1 indicates the start of the execution of T_1 ; the firing of T''_1 indicates the end of the execution of T_1 ; and a token in place P' indicates that transition T_1 is executing. We use the representation given in Figure 2a for our computational models. The parameters associated with transition T_i are as follows:

e_i = random variable representing the time at which T_i is enabled,

s_i = random variable representing the time at which T_i commences execution,

f_i = random variable representing the time at which T_i finishes execution,

$e_i - e_i$ = waiting time for T_i ,

$e_i - NST$ = latency time for T_i , where NST is network start time.

It is appropriate to note that s_i and f_i represent the firing of the two virtual transitions associated with transition T_i and that the expected value of $f_i - e_i$ is the input parameter E_i , the average execution time of T_i . Latency is defined as the time that a transition must wait before it is enabled.

EVENTS

For PARSIM we have four types of events. They are as follows:

Start_Simulation (type 1)

End_Simulation (type 2)

Transition_Commences_Firing (type 3)

Transition_Ends_Firing (type 4)

The simulator is "event-driven"; i.e., the simulation proceeds by executing the state changes as specified by the events in the Event-Table. The Event-Table is a dynamic table of rather simple structure. It contains a list of the times at which an event will occur, the type and classification of event involved, and an identifier of the transition associated with the event. Events are classified as either primary (P) or conditional (C); primary events may cause the creation of new future events independent of the state of the system, while conditional events may cause new event creation based upon the system state. A typical event table is shown in Table 1.

The events to be added to the table are determined by the event-time prediction routines. These routines select events (transition firings) based upon the set of enabled transitions and the particular rules of a given computation. For example, when several transitions are enabled, the next transition to be fired determines an event which is to be placed in the event table. That transition is selected according to rules as decided by the user's interpretation of the net. The selection may be made at random or by some other rule.

In PARSIM, type 3 events are primary events since we always have a future type 4 event associated with the occurrence of a type 3 event. The event-time prediction routines use the "execution discipline" or set of execution rules which is as follows.

Execution Discipline

1- No two events can occur at the same time. The minimum time between the occurrence of a pair of events is controlled by the value of the input parameter $Min_Time_Between$.

2- Transitions can fire when enabled.

3- A transition which has fired cannot be scheduled for a subsequent firing until after its scheduled end firing. The virtual transition and the virtual place are safe; i.e., can have at most one token.

Simulation Procedure

1 Start the simulation, read input data, initialize.

2 Select event from the event table.

3 Advance clock to designated time.

4 If primary event create new events add to event table.

5 Change Status Description.

- 6 Execute conditional events routines.
- 7 Add new events to event table.
- 8 Record data and go to step 2.

Set of PARSIM Procedures

- 1 Main program to control the flow of the simulation.
- 2 Random number generators.
- 3 Statistical routines for preparing and processing output table.
- 4 Routine for reduction of the number of transitions firing in parallel. (This routine will be put in abeyance initially. It can be determined within the context of a Petri net.)
- 5 Set of enabled transitions routine.
- 6 State change routine. Upon passing of tokens what is the next state.
- 7 Event - time prediction routines. These analyze the current state and use the state change routines to give the time and type of next state.
- 8 Routine for selecting a transition from the enabled set according to the network discipline.
- 9 Routine for selecting the next simulation event.
- 10 Routine to process event table.
- 11 No progress routines. These detect deadlock and also are used to detect infinite unplanned looping.
- 12 Input/Output interface.

5. USER INTERFACE

A preliminary prototype interface, which used alphanumeric input only, was developed for the description of the net to be simulated. This interface includes inputs for the following.

1. Two matrices D^+ and D^- are used to define the topology of a given net. The dimensions of the matrices are N_t by N_p representing the number of transitions and places, respectively.
2. An initial state representation where state is defined as follows.

$STATE(i) = (state1, \dots, stateN_p) \cup BUSY,$
where $statej$ represents the number of tokens in the place j at time t .

$BUSY$ is a binary vector where, for i in $1..N_p$,

$$BUSY(i) = 1, \text{ if transition } i \text{ is executing at time } t; \\ = 0, \text{ otherwise.}$$

- 3 Transition execution time is in a real array called $Expected_Times$ of length N_t .
4. A parameter to control the maximum times the system can transverse through a loop.

5. A parameter to set the minimum time between the firing of transitions.

6. An initial seed for the random number generators.

In addition, the interface contains an on-line help facility which provides basic information regarding the operation of the simulator. The preliminary prototype interface was developed originally for a restricted set of users; i.e., students in a graduate real-time systems class. Subsequent work includes emphasis on development of a graphics based "friendly" interface for a more general set of users. The graphics based interface allows for provision of input interactively or via data files.

PARSIM allows users to view output interactively or to route output to any of the output devices. The output includes a statistical packages which give statistics on the following net parameters.

1. Transition latency times.
2. Transition waiting times.
3. Transition execution times and firing frequencies.
4. Place- token distribution.
5. Total net execution time.
6. Reachability tree.

The interface allows, in addition, a "debug" mode. In this mode, a user can print out tables such as the state table at each time step, the event or transition table thus allowing detailed analysis of a simulation design.

PARSIM was tested on a number of small to medium sized Petri-nets ranging from somewhat simple to rather complex topologies. For seven of the networks run on PARSIM, we present the resulting data in Table 2.

The column labeled TOKENS represents the number of tokens initially in the network. The times in seconds represent the total real time for the simulator to complete execution. Execution was completed by arriving at a state in which there were no possible additional transition firing; or having the looping limit reached. That is, the network had repeated a sequence of transition firings up to a limit set by an input parameter. This was the case for net number 6, which displays the largest total execution time of 2.80 seconds. Network number 6 had the most complex network topology. These times are for the AT&T PC 6310, with 640K ram, a 20 megabyte hard disk, and a 1.2 megabyte floppy drive. The times given do not include the input times for describing the net topologies. The preliminary version of PARSIM consisted of approximately 1500 lines of Ada code run on the Meridian AdaVantage(un) Compiler version 2.0.

The current implementation of PARSIM is being ported to an AT&T 3B2/500 Unix Ada environment. The focus is on the enhancement of the user interface and the expansion of the size of Petri-nets for which the tool may be applied. A practical Markovian extension to the simulator is part of the enhancements in the Unix environment. The Markovian version is based on the representing the Petri-net as a Markov process and the solution of the resulting equilibrium equations. PARSIM will be used as a tool for the analysis of real-time concurrent software. We intend to use PARSIM in our research on software reliability in real-time systems and to measure the overhead of redundancy checks used in reliable real-time software systems.

7. Summary

This paper has presented an overview of a Petri-net tool for the simulation of parallel and real-time systems. The simulator has been used to model complex but small (transitions) nets. PARSIM provides comprehensive statistics on the execution of a Petri-net. A variety of options are available for user input and output. The present version runs on an IBM compatible PC with at least 640 KB memory.

8. Acknowledgements

The authors gratefully acknowledge support from the Department of Defense under contract # MDA904-88-C-4169. The second author also gratefully acknowledges support from the Army Research Office under contract # DAAL-06-G-0085. Both of the authors are particularly indebted to the following Real-Time Systems graduate students at Howard University who developed some of the software for PARSIM: Robert Bennett, Marcus Reed, Marvin Bingham, and Ivan Brooks.

9. References

- Cherry, G., Parallel Programming in ANSI Standard Ada, Reston Publishing Company, 1984.
- Holliday, M. and Vernon, M., "A Generalized Timed Petri Net Model For Performance Analysis", IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, December 1988, pp 1297 - 1310.
- Peterson, J., Petri Net Theory and Modeling Of Systems, Prentice-Hall, 1981.

DON MICHAEL COLEMAN is a native of Detroit, Michigan. He received the B.S., M.S.E., and Ph. D. degrees in electrical engineering from the University of Michigan. From 1960 to 1964, Dr. Coleman worked as a systems engineer at the A.C. Spark Plug Division of General Motors, and from 1965 to 1971 at the Institute for Science and Technology and Systems Engineering Labora-

tory at the University of Michigan. Currently, Dr. Coleman chairs the Department of Systems and Computer Science at Howard University where he has been since 1971. His research and professional interests center around software engineering, fault-tolerant computing, systems engineering and technology and development. He is a member of the IEEE and the Association for Computing Machinery.

RONALD J. LEACH received the B.S., M.A., and the PhD in Mathematics from the University of Maryland at College Park and a M.S. in Computer Science from Johns Hopkins University. He is a Professor in the Department of Systems and Computer Science at Howard University where he has been teaching since 1969. His research interest include software engineering and software metrics, fault-tolerant computing, computer graphics and analysis of algorithms. He is a member of the Association for Computing Machinery, IEEE Computer Society, and the Mathematical Association of America.

Time	Event	Identifier	Class
234	type 1	0	C
235	type 3	6	P
247	type 4	6	C
238	type 3	4	P

TABLE 1: A TYPICAL EVENT-TABLE

PLACES	TOKENS	TRANSITIONS	ARCS	TIMES
2	6	1	2	0.109
3	5	2	5	0.159
5	3	4	16	0.989
6	12	3	8	0.279
7	13	3	11	0.219
15	8	10	40	2.800
25	11	18	66	0.140

TABLE 2: RESULTS



FIGURE 2a Generic Transition

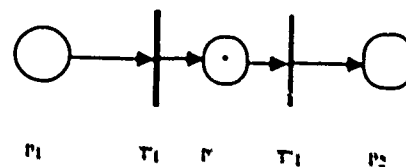


FIGURE 2b Executing Transition

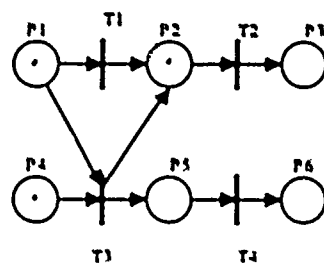


FIGURE 1a

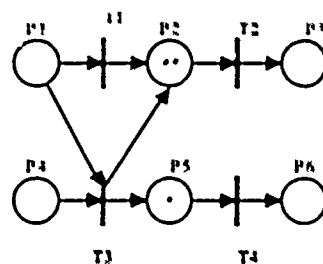


FIGURE 1b

Real-Time Pattern Recognition in Ada: On the Formulation of Neural Net Recognizers
by Ada Tasking of Massively Parallel Multicomputers

William Arden

Telos Federal Systems

ABSTRACT

This paper examines findings in the technologies of Neural Nets, Ada-Tasking, and Parallel Distributed Processing (PDP) multicomputers for their integrated approach to Real-time Pattern Recognition. This approach involves formulating Neural Nets in terms of Ada-Tasking and examining the Ada-Tasking model for networked VLSI microcomputers. The result is then the class of pattern recognizers based upon Neural Nets which may be coded into Ada and distributed onto host multicomputer architectures. The primary benefits of this paper are in the relation of the now developed technologies of PDP multicomputer architectures and neural networks to the Ada application area of Real-time C-I pattern recognition.

INTRODUCTION

Lessons are being learned in three distinct technologies based upon a common theme with powerful application in the IEW component of C-I systems: Ada Language with its Tasking Capability, Multicomputer Chip Sets, and Neural Nets. The common theme for all three technologies is their enormous potential with PDP.

This paper is an analysis of lessons learned in PDP with these differing technologies in regard to their utility for Real-time (Subliminal) Pattern Recognition. This means that the process must complete below (sub) the threshold of perception (the limen). This threshold may be set in the centi-second range; below this range events are not detectable by human observers unassisted by instrumentation, (for example, consider what is meant by "the hand is quicker than the eye").

As for Pattern Recognition, we are speaking of the identification of objects by inputs that may be distinguished by pattern (i.e., characteristics, attributes, structure, geometry, topology, algebra, etc.). For example, if track data were to be supplied for ICBMs in boost phase versus decoys with similar trajectories, the sensor track data could be interpreted within centi-seconds to make the identification (in this case also the decision) real or decoy. This would then be Real-time Pattern Recognition.

Neural Nets, which are fashioned after brain circuits, are a powerful technology of pattern recognition. Due to an inherent parallel and distributed processing nature, these nets also offer a tremendous capability for Real-time performance.

This author has been researching a common logical interface between Neural Nets and PDP Multicomputers. In this regard Ada-Tasking constructs supported by a PDP Multicomputer Ada Compilation System (ACS) are ideal for flexible Real-time systems. The common logical interface is demonstrated by representation of Neural Nets, Ada-Tasking, and the PDP Multicomputer Ada model by extended Petri Nets.

This implementation strategy of Ada Neural Nets for PDP is the material to follow. Such Ada Neural Nets assigned to a PDP Multicomputer offer tremendous potential for Real-time pattern recognition and are yet flexible (not resorting to numerous costly hardware modifications) and portable. This is due to the implementation strategy of using Ada-Tasking Software for Neural Net computation.

Neural Nets are now provably the best pattern recognizers for patterns above modest complexity. Researchers at EEG Systems Laboratory, a Government sanctioned research institute in San Francisco, California, have stated "we compared the (Neural) Network with standard statistical tests and the network was better". DARPA has recently released a 600 page report entitled the "DARPA Neural Network Study" which stresses the importance of Neural Networks as computational structures which adapt and learn. The Lincoln Laboratory of MIT has reported the feasibility of Neural Nets in tracking the Stealth aircraft, and DARPA assistant director Jasper Lupo has stated that neural nets are going to be "more important than the atom bomb". The importance of Neural Net technology to Pattern Recognition should not be underestimated.

Currently, the models of cognitive systems by which Neural Nets are formulated are being done formally as math models. There is a general class of cognitive systems, including Boltzman machines and harmonium graphs, described as the quintuple.

$$C = (R, P, O, \lambda, C)$$

This is described by an example of a harmonium graph under its interpretation as an extended Petri Net in Appendix A. In fact, it has been shown that there is a method of translating these cognitive systems into extended Petri Nets. These extended Nets have the extensions of inhibitor arcs, colored tokens (red and blue), and a stochastic firing rule. This is depicted in Figure 1. The power of the Petri Net Representations (PNRs) is that Ada-Tasking models for the Multicomputer may also be represented similarly.

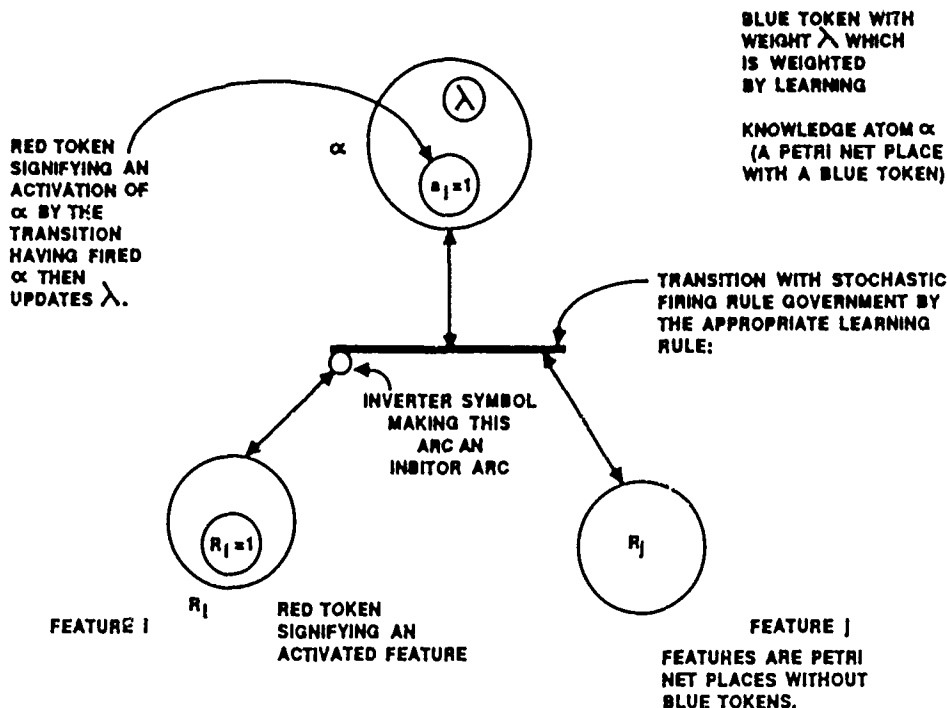


FIGURE 1. AN EXTENDED PETRI NET REPRESENTATION OF A SIMPLIFIED COGNITIVE SYSTEM (NEURAL NET)

2.0 ON Ada-TASKING MODELS OF NEURAL NETS

We will begin by examining the simple PNR of a Neural Net given in Figure 2. This PNR uses bidirectional arcs in place of its equivalent in input and output arcs, places, and transitions. (This example is taken from the more formal example given in Appendix A.)

We will speak of places and transitions as PNR elements which are linked by input/output arcs. The Ada-Tasking model of a PNR Neural Net is simply the correspondence of an Ada Task for each of the PNR elements. The PNR Neural Net is such that all of its elements perform processing in a parallel and distributed fashion. In this regard, the tasks which correspond to each element may also execute in a parallel and distributed manner.

The following three examples of Ada-Tasking generally explain the nature of the Ada Tasks involved and

illustrate how Ada-Tasking may be formulated for each of the PNR Neural Net elements:

Example 1: This task corresponds to place Panther in Figure 2.

```
task Panther_Recognition is
  entry Transition_one (...);
end Panther_Recognition;
task body Panther_Recognition is
  -- declaration of the
  activation indicator
  boolean variable
begin
  loop accept Transition_one
    (...) do
    -- set the activation indi
    cator to one
    -- if recognition of a panther
    occurs
    -- otherwise set it to zero.
    end Transition_one;
    exit when -- recognition occurs
  end loop
end Panther_Recognition;
-- we would do task Top_Hat_
Recognition similarly
```

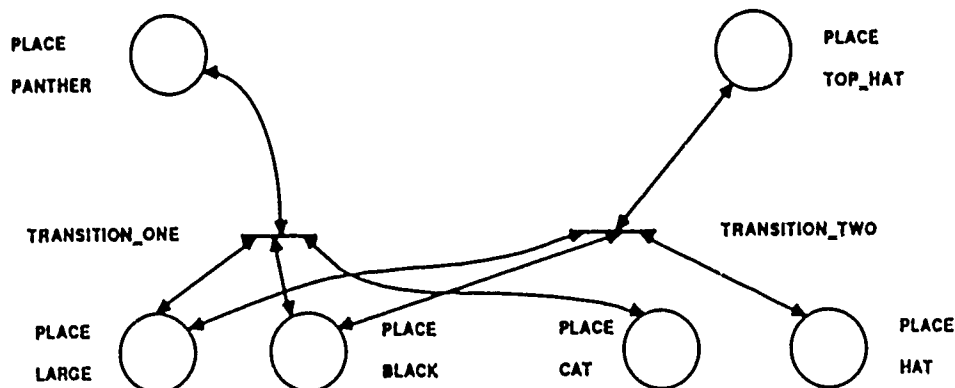


FIGURE 2. A SIMPLE PETRI NET REPRESENTATION OF A NEURAL NET

Example 2: This task corresponds to the transition named Transition_one in Figure 2.

```

task Transition_one is
  entry large (...);
  entry black (...);
  entry cat (...);
end Transition_one

task body Transition_one is
-- declare boolean indicator

begin
  loop accept large (...) do
-- perform neural net computa
tion
-- on inputed value fire tran
sition
-- if appropriate end large;
  accept black (...) do
-- process as above
end black;
accept cat (...) do

  -- process as above
end cat;
  exit when -- desired
recognition occurs
end loop;
end Transition_one;

-- task Transition_two is done
similarly

```

Example 3: This task corresponds to place large in Figure 2.

```

Task large is
-- appropriate entry statements
end large;

task body large is
-- if we refer to the example
in Appendix A
-- task large is for a place
-- which is a knowledge atom
-- and so we would declare
-- a variable for and
-- a boolean variable for a;

begin
  loop
-- use appropriate accept
statements
-- compute by a learning
rule
  exit when -- recognition or
not occurs

  end loop;
end large;
-- task black, cat and hat
-- could all be done similarly

```

It should be mentioned that Neural Nets (or cognitive systems) have larger scale architectures which may also be given as PNRs. These larger scale architectures are called Schema, and a top-level representation of a Neural Net is said to be a Schematic representation. Each of the PNR elements in a Schema may be substituted for by more elaborate Neural Nets. This is illustrated in Figure 3.

Here a top-level cognitive structure called a Schema is given. Each of its places and transitions can be filled in by substituting other nets. In this way, a top-level design of a Neural Net may occur.

Similarly, Ada Tasks may correspond to Schema PNR elements by incorporating the necessary sub-net PNR element associated tasks into the Schema PNR element task. A template for the task corresponding to the substituted net in Figure 3 is given in the following example:

```

task substituted_net is
  task transition_one is
    task place_one is
      o
      o
      o
    end transition_one
  end substituted_net
task body substituted_net is
  task body transition_one is
    task body place_one is
      o
      o
      o
    end transition_one
  end substituted_net
begin
  loop
    substituted_net;
    transition_one;
    place_one;
  end loop;
end substituted_net;

```

This task, which is comprised of all the tasks corresponding to each of the places and transitions of the substituted net, will correspond to the transition in the Schema Net of Figure 3.

Since Ada Tasks may be incorporated into other Ada Tasks as shown above, the Ada Tasks may be sized to the appropriate grain (be it fine or coarse grain) of the Multicomputer architecture.

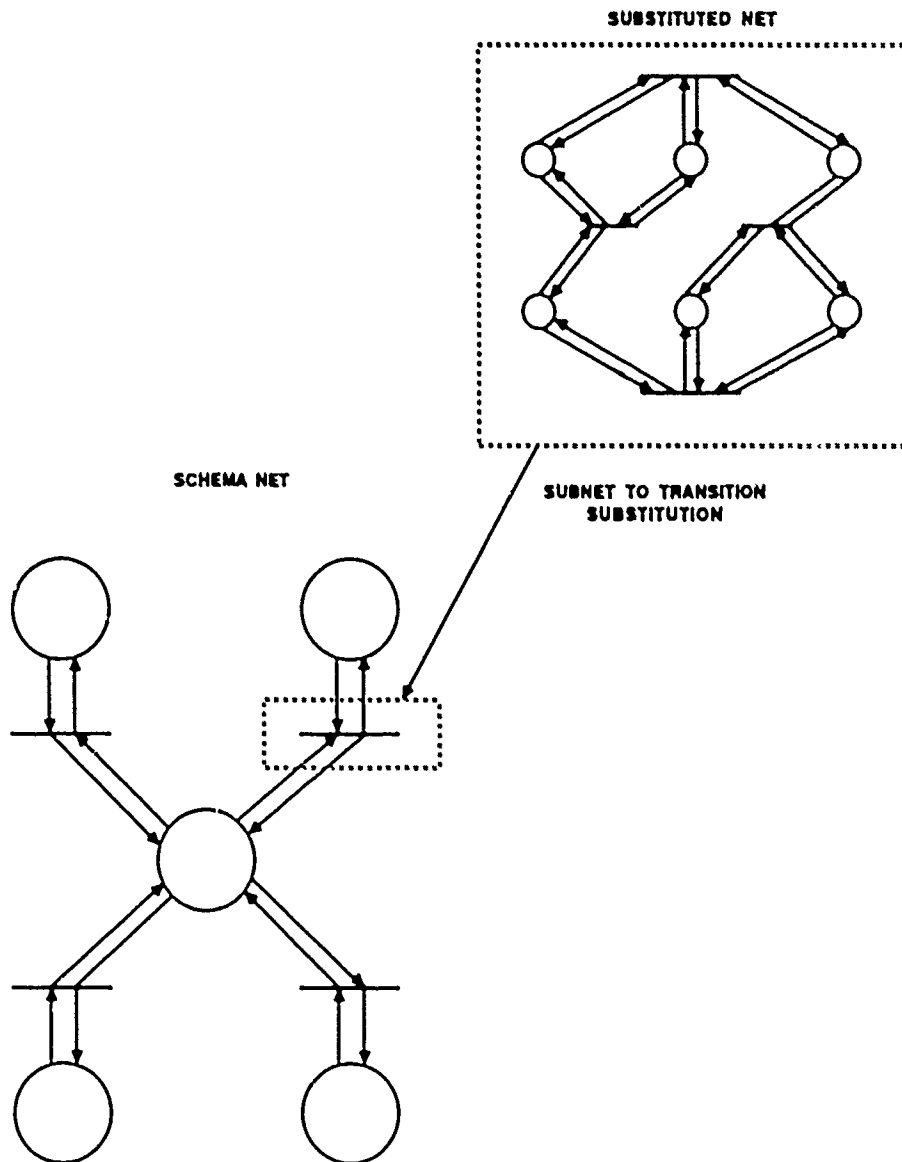


FIGURE 3. SCHEMA SUBSTITUTION

3.0 ON Ada COMPILATION SYSTEMS FOR PARALLEL DISTRIBUTED MULTICOMPUTERS

Naturally, Ada-Tasking for Neural Net Recognizers needs to be compiled for and bound (linked) on a computer by computer basis for multi-computer architectures.

Lessons learned in this arena are primarily from ALSYS Ltd., in joint endeavor with Inmos, to develop an ACS for the Inmos VLSI microcomputer named the Transputer. The Transputers have included:

- 1) The lms T414: 32 bit processor, operating at 10MIPS, 2K on-chip RAM, memory addressing to 4 gigabytes and 4 links each providing 20Mbits DMA communications rate.
- 2) The lms T212: 16 bit processor operating at 12 MIPS, with 4 links.
- 3) The T800: 32 bit processor and 64 bit floating point unit exceeding 1 Mflop, 4 K bytes on chip RAM.

These Transputers may be linked point to point or via a "telephone exchange" of custom silicon for such purposes as an application specific topology.

The application specific topology for the class of pattern recognizers (Ada-Tasking Neural Nets) in this paper would tend to be a n-ary tree architecture. In most cases, point to point links alone would suffice.

The Transputer systems commercially available include:

- 1) The Floating Point Systems T-series of supercomputers operating at 128 Mflops for \$0.5m
- 2) The Meiko Computing Surface operating at 2 Giga Instructions per second.

Each of these systems adopt a network strategy for the VLSI microcomputers (Transputers), resulting in parallel distributed processing multicomputers.

ALSYS Ltd. is also developing an ACS for such networked Transputers. The compilation system is to consist of:

- 1) Ada Library Management Utilities
- 2) Ada Compiler

3) Ada Binder

4) Ada Runtime System.

The ACS is being formulated into an Ada model wherein each Ada Task is to be represented by a transputer process. The choice of which Ada Task to schedule at any time is made by the Ada Run-time Kernel.

The Ada model of the ACS is ideal for implementing Ada Tasks based upon the PNR elements discussed in this paper. This would allow for the potential of massively parallel distributed processing for Neural Net computations. The reason that Neural Net implementation on a networked Transputer multicomputer is ideal is that Ada-Tasking becomes a common logical interface between the Neural Net PNRs (as shown) and the Transputer network (by virtue of the Ada Task Transputer process model of the ACS).

4.0 CONCLUSION

The preceding has been an analysis of established results. The analysis herein clearly indicates the potential for Real-time pattern recognizers which are implemented via Ada-Tasking on PDP multicomputers. The advantages of the Ada tasking approach include the flexibility of modifications to the Neural Nets and the capability of porting such Ada-Tasking implemented recognizers to other hardware platforms.

ACKNOWLEDGMENTS

I wish to acknowledge JCF Barnes for his correspondence on the Ada Compilation System for Transputer Networks.

REFERENCES

1. "Neural Networks at Work", Scientific American, Nov. 1988, pp. 134-135.
2. "DARPA Neural Network Study", Signal Magazine, Nov. 1988, pp. 8.
3. "Neural Nets Could Track Stealth Aircraft", Advanced Military Computing, Aug-Sept. 1988, pp. 5.
4. Same as 1.
5. "The Analysis and Synthesis of Intelligent Systems" William Arden, May 1988.

6. "Parallel Distributed Processing, Explorations in the Microstructure of Cognition", J. McClelland, D. Rumelhart and the PDP Research Group, Vol. 1 MIT Press 1986.

ABOUT THE AUTHOR

Mr. Arden is a senior engineer and task leader for Ada technology research at Telos Federal Systems. His experience includes research into pattern recognition, AI and neural networks. He has conducted math-modeling and simulation in support of sensor recognition systems. Mr. Arden holds a B.A. in mathematics from Stockton State University and a M.S. in Electronic Engineering from Monmouth College.

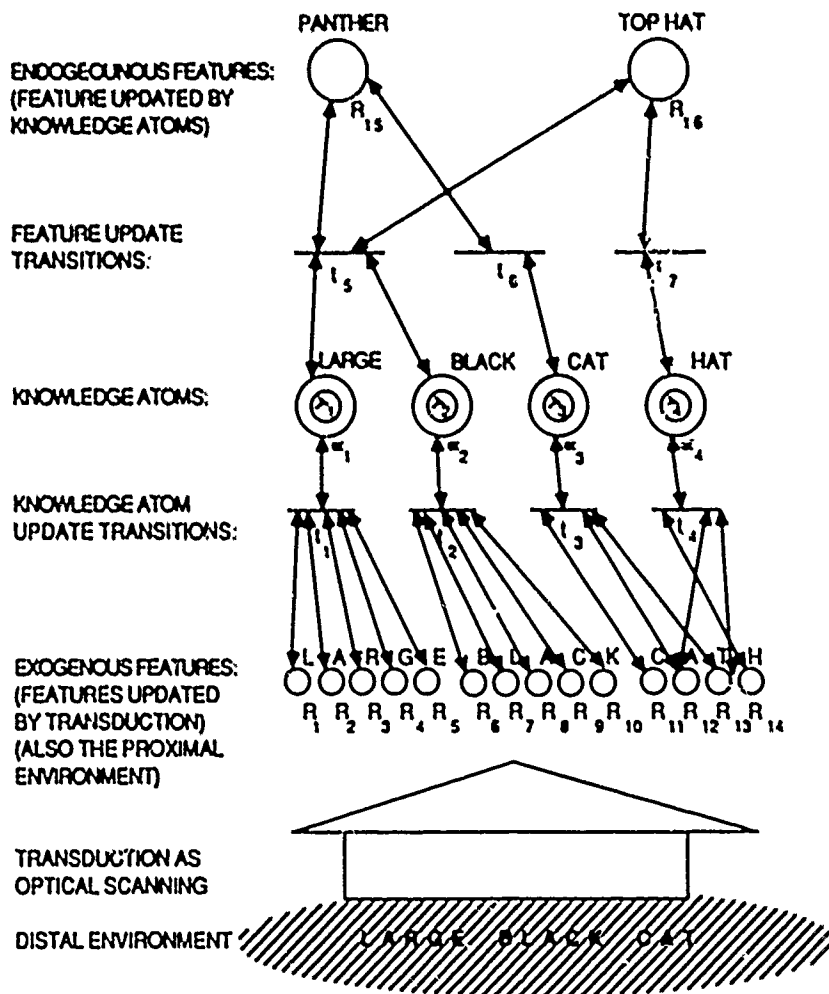


FIGURE 4.

APPENDIX A

Here we will give an example of a simple cognitive system represented in Petri Net form. This cognitive system is represented in Figure 4.

We will discuss it first as a cognitive system or C. Given our definition of a C, as the quintuple: (R, P, O, π, C) , we see from the figure that:

$$R = \{R_{ex}; R_{en}\} = \{R_1, R_2, \dots, R_{14}, R_{15}, R_{16}\} \\ \{R_i\} = \{R_1, R_2, \dots, R_{16}\}$$

P = the Proximal environment is the probability distribution of the activation of $R_1 \dots R_{14}$ by the optical scanner.

O = the set of observables is here {large, black, cat, hat} given by connections K_α and knowledge atoms α .

Before we go on and show π and C we will first mention that choices of π and C will specify types of cognitive systems and vice versa. Here, we will choose a harmonium machine described in the PDP series Volume 1 [47].

For the harmonium machine we have:

$$C \text{ such that: probability } (a_i = 1) = \frac{1}{1 + e^{-2/T}} \\ \text{and} \\ \text{probability } (R_i = 1) = \frac{1}{1 + e^{-2/T}}$$

where $a_i = 1$ is a knowledge atom activation and $R_i = 1$ is a feature activation and T is a parameter called the Computational Temperature usually set at $T = 1$ and lowered to force decisions.

Also here:

$$I_\alpha = \sum W_{i\alpha} \cdot r_i - \lambda_\alpha \\ \text{and} \\ J_i = 2 \cdot \sum W_{i\alpha} \cdot a_\alpha \\ \text{and} \\ W_{i\alpha} = (K_\alpha)_i \frac{\lambda_\alpha}{|K_\alpha|} \text{ where } K \text{ is a proportionality constant to be chosen as } 1 > K > 1 - 2/(\max_\alpha |K_\alpha|)$$

Here the subscript α denotes a particular knowledge atom and R_i a particular feature.

This C is chosen because it maximizes $H(r, a)$ for harmonium machines.

where H is the harmony between features updates and knowledge atom activations. We will merely state $H(r, a)$ as:

$$H(r, a) = \sum_\alpha K_\alpha \lambda_\alpha h(r, K) \\ \text{and} \\ h(r, K_\alpha) = \frac{r \cdot K}{|K_\alpha|} - K$$

where K is the constant seen before: $1 > K > 1 - 2/(\max_\alpha |K_\alpha|)$.

We will not go into the details of this formulation as it diverts us from the intentions of our example. The interested reader should see the chapter on Harmony Theory in the PDP series [47]. Let us simply state that the completion function given will serve to maximize.

We have now specified the C = (R, P, O, π, C) . Let us return to the figure and see how it behaves as a CHN. Firstly, red tokens will appear with probability P on our places $R_1 \dots R_{14}$ with each pass of the optical scanner. These are the feature activations. Our transitions $t_1 \dots t_4$ will fire according to the completion function for knowledge atom updates. Similar transitions $t_5 \dots t_7$ will fire according to the completion function for feature updates. For several passes of the optical scanner the blue tokens of the knowledge atoms will begin to weight by λ . In fact, this CHN can be trained by weighting λ_3 for the cursive writing giving the word cat. This CHN will then correctly decide Panther instead of Top Hat and recognition will have been performed.

What we notice from this example is that the PNR readily models the C. Beyond this, the C in PNR form is now capable of undergoing the powerful techniques of Petri Net Synthesis and Analysis.

**TASKIT:
AN Ada SIMULATION TOOL KIT FEATURING MACHINE INDEPENDENT
PARALLEL PROCESSING**

Michael Angel and Paul Juozitis

General Dynamics - Data Systems Division

Abstract

The Tasking Ada Simulation Kit (TASKIT) is a set of software tools that enables model builders to develop simulations in Ada. The tools are implemented as user-callable procedures and functions and provide simulation services common to many types of modeling applications. TASKIT uses an activity oriented modeling approach that is a variant of discrete event modeling. An important feature of TASKIT is that it enables simulations to take advantage of tightly coupled parallel processing. This parallel processing capability, which is completely machine independent, has been successfully tested and benchmarked on a parallel processing machine. TASKIT was developed under contract to the Software Technology for Adaptable Reliable Systems (STARS) Program, in conjunction with the Naval Research Laboratory, and is available through the STARS Ada Foundations software repository.

Introduction

Discrete event simulations that require detailed and sophisticated algorithms often suffer from slow execution times. This is especially true for simulations that have a large number of entities. Consequently, the model builder often must sacrifice detail in the interest of faster execution time. Parallel processing holds great promise for these large and computation-intensive modeling applications. Distributing the processing for concurrent simulation events could greatly reduce the execution time for a simulation. This performance improvement would allow the model builder to increase the level of detail or the number of entities in a simulation if desired, as well as reduce the turn around time for producing simulation results.

TASKIT provides this parallel processing capability for simulations. It takes advantage of the Ada tasking mechanism and is designed to run on any tightly coupled parallel processing machine that supports concurrent Ada. Tightly coupled parallel processing allows processors to share memory, which is an

important requirement for TASKIT. Several machines of this type in the commercial marketplace support concurrent Ada. Loosely coupled parallel processing, where processors do not share memory, is not addressed in this paper and is not supported by TASKIT.

TASKIT was designed with two key goals in mind. First and foremost, the tools had to be machine independent, including the parallel processing capability. The idea was to design the software so that it could be easily ported to different machines, and so that future advances in hardware technology would not make the software obsolete. The second goal was to make the parallel processing feature optional to ensure that the simulation tools were efficient in both sequential and parallel environments. Both goals were achieved.

Ada as a Simulation Language

The simulation services that TASKIT provides, combined with the inherent features of the Ada programming language, transform Ada into a robust simulation language. The use of a standard programming language with appropriate support routines to write simulations is not a new idea. This has already been done in several high level languages. However, Ada offers many advantages over other high level languages for the development of simulation models.

One advantage is that the standardization of Ada is much more rigid than other existing languages. Ada compilers are required to pass a suite of validation tests before they are fully accredited. The result of this rigid enforcement of standards is portability. A simulation written using TASKIT and Ada can easily be ported to another system without modification.

Another advantage is that Ada promotes structure and readability. As a result, simulations constructed in Ada are easy to maintain. Maintainability is key for many simulations since models often have a long lifespan and are frequently changed and enhanced.

Figure 1 shows an example of a call to a TASKIT procedure to create a simulation activity. The code in this example is very readable. The readability is greatly enhanced by the Ada feature that allows the explicit naming of arguments in a procedure call.

```

Create_Activity
(ACTIVITY_CLASS => AIRCRAFT_MOVE,
 START_TIME    => 0.0,
 END_TIME      => 100.0,
 TIME_STEP     => 10.0,
 ACTIVITY_DATA => AIRCRAFT_MOVE_ACTIVITY_DATA);

```

Figure 1 - Example of a call to a TASKIT procedure

In addition, Ada provides features that make simulation tools written in Ada easy to use. For example, an Ada procedure can be overloaded and can contain default values for its arguments. This feature enables TASKIT to provide a great deal of power for the sophisticated user, without overwhelming the novice user. Other Ada features that promote ease of use include packages, information hiding, and strong typing. These features enable the model builder to use an object oriented approach for constructing simulations.

Finally, the major advantage Ada offers is the task, which is a virtual unit of concurrency. The task is a very powerful feature of Ada because it allows for machine independent parallel processing. Parallel processing hardware that can exploit the use of the Ada task exists today. The combination of parallel computers and Ada tasking holds great potential for portable parallel processing applications and tools. TASKIT is one example of a tool that has tapped this potential.

The Activity Orientation

The fundamental concept behind TASKIT is that any system can be modeled in terms of activities. An activity has a user-specified start and end time, and a user-written Ada procedure that contains the algorithms used to model a real world activity. Each activity also has a unique user-specified time step associated with it so that the activity duration can be subdivided into small discrete time steps. Once an activity is started, TASKIT calls the user's procedure at each time step until the end of the activity. A data record is passed to the activity procedure at each time step to provide data local to the activity. The structure of an activity is flexible and allows the user to turn the time step mechanism on and off. When the time step is turned off, an activity will only be called once at the start and once at the end.

This activity orientation is a variant of discrete event modeling. Essentially, an activity is a construct that combines a series of regularly occurring discrete events to represent a continuous process. In discrete event simulation, an activity can be represented by an event that reschedules itself at a fixed time increment. Conversely, an activity can be used to represent a discrete event by setting the start time equal to the end time and setting the time step to zero.

The primary reason for adopting the activity orientation in TASKIT was to set up a simulation methodology conducive to parallel processing. Instantaneous discrete events do not conjure up a notion of concurrency. Activities however, can conceptually be thought of as executing in parallel with other activities, since an activity has a duration and is not instantaneous. TASKIT takes the conceptual notion of parallel activities and turns it into a reality through the use of the Ada tasking mechanism.

For each activity created by the user, an Ada task is created by TASKIT. The call to the user's activity procedure is embedded within the task that is created. It is important to note that the user does not create Ada tasks, only procedures. It is TASKIT's job to map the activity procedures to an Ada tasking environment. Since TASKIT creates and manages the Ada tasks, the tasking feature is optional. If parallel hardware is unavailable, the user can turn off the tasking feature to avoid the additional overhead associated with tasks.

Activities and Parallel Processing

There are some rules as to when activities can take advantage of parallel processing. The first rule is that TASKIT can only execute activities in parallel if their corresponding time steps occur at the same point in simulated time. This is not an unreasonable assumption. The alternative is to allow a situation where one activity could be farther ahead in simulated time than another activity, although both are executing concurrently. Letting activities get ahead of one another can lead to problems, especially if a slower activity causes some change in the system that affects a faster activity somewhere ahead in the future. That is not to say it is impossible to handle this type of situation. Rather, it is much less complex and more intuitive to only allow activities that occur at the same time to execute concurrently. TASKIT provides a method for synchronizing activities to maximize the amount of parallelism.

The second rule for parallel activities involves the concept of an activity class. A class consists of one or more identical activities that may or may not operate on the same data. In TASKIT, activities that belong to the

same class can execute concurrently. The model builder specifies whether or not the activities in a given class are to execute in parallel. In addition to having the capability for activities within a class to execute in parallel, it is also possible for different classes to execute concurrently. For two or more classes to execute in parallel, they must have the same user-specified priority. The reason for the priority is to handle the case where one class of activities may depend upon the results of another activity class, even though they occur at the same time. Figure 2 illustrates the concept of activities within a class, as well as multiple classes executing concurrently.

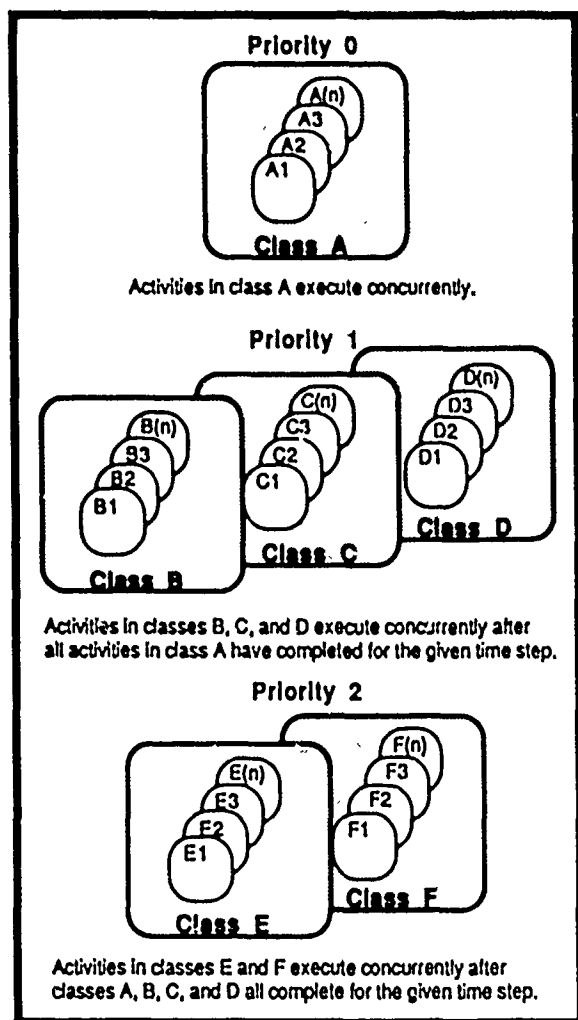


Figure 2 - Activities of the same class execute concurrently and activity classes of the same priority execute concurrently

The third rule for parallel activities pertains to the use of global data. On a tightly coupled parallel computer, shared memory allows parallel activities to access shared global data. The user must take certain precautions when using global data in a parallel environment. Read operations on global data require no special treatment for parallel activities. However, read-write operations performed on the same global object by parallel activities can cause problems. TASKIT provides a semaphore package so that the model builder can restrict the access to certain areas of global data. A semaphore simply queues up requests to a critical code section where access to a restricted resource occurs. The resource can be an area of global data, a file, or an external device. Semaphores are necessary in some cases, but their use should be minimized. A resource protected by a semaphore causes parallel activities requiring that resource to remain in a wait state until access is obtained. This results in a loss of parallelism.

Following the rules described above will not guarantee that a simulation with parallel activities will run faster on a parallel computer than on a sequential computer. Parallel processing is more like an investment than a gift. An investment must be made before profit is realized. In the TASKIT simulation environment, the investment is the processing overhead required to manage the Ada tasks at the TASKIT tool level and at the Ada run time level for a particular machine. The profit is an overall reduction in computation time. In order to realize a profit, the time saved by distributing computation must be greater than the time spent managing tasks.

To realize a computational profit, a TASKIT simulation must have parallel activities that contain sufficient granularity. Granularity denotes the duration of processing between synchronization points in a task. It is a measure of the amount of work to be performed by a particular activity. For example, an activity that represents a radar site trying to detect an incoming missile would have a low granularity if the activity only performed a simple range check for radar detection. If the activity used a sophisticated radar detection algorithm, it would have a much higher granularity. A set of parallel activities with very little computation does not have sufficient granularity to justify an investment in parallel processing, while a computation-intensive set of parallel activities should realize a handsome profit.

In addition to having parallel activities of sufficient granularity, the application should also have a suitable number of parallel activities. The model builder has control over the number of activities within a class and the number of activity classes. The number of parallel activities should map closely to the number of

processors. Fewer concurrent activities than processors results in unused processors, while more concurrent activities than processors results in extra context switching.

Benchmark Results For TASKIT on a Parallel Machine

Benchmark experiments were constructed to validate TASKIT in a parallel processing environment. Two goals were set when designing the experiments for the benchmark. The first goal was to demonstrate that TASKIT can be ported to a parallel processing machine without requiring modification. The second goal was to show that simulations using TASKIT on a parallel machine can experience performance improvements under the appropriate conditions. Both goals were achieved.

A test simulation was constructed using the TASKIT tools and then ported to an Encore Multimax™. The Encore is a tightly coupled parallel processing machine that supports a validated concurrent Ada compiler. The Encore used for the benchmark contained 12 processors although the machine was capable of utilizing up to 20 processors.

The first goal of demonstrating portability was met by successfully compiling the test simulation and the TASKIT software on the Encore. No modification was required for TASKIT to take advantage of the parallel processing capability of the Encore. This was a significant achievement since TASKIT was developed on a sequential machine, a VAX™ 11/780. The TASKIT development team did not have access to any parallel processing hardware when TASKIT was being constructed. TASKIT was developed using a "software first" approach. This approach emphasizes machine independence and more importantly, reverses the natural tendency to develop software specifically for existing or available hardware.

After successfully rehosting TASKIT to the Encore, the second goal of the benchmark was to prove that TASKIT simulations could benefit from parallel processing under appropriate conditions. As discussed earlier, only activities of the same priority that occur at the same point in simulated time can be executed concurrently. For this benchmark, all activities in the test simulation belong to a single class and therefore have the same priority. Additionally, all activities are synchronized to the same time step. This represents an ideal case. (An actual simulation might require the activity classes to have different priorities or time steps.)

Multimax is a trademark of the Encore Computer Corporation
VAX is a trademark of the Digital Equipment Corporation

The work performed by the activities in the benchmark consists of a floating point addition that operates on local activity data. This addition operation was placed inside a loop so that various levels of granularity could be measured.

The benchmark consisted of three main cases for the test simulation. The first case contained only one activity, the second case five activities, and the third case ten activities. For each case, the test simulation was executed for 1000 simulation seconds with a time step of 10 seconds. The granularity was parametrized with values of 0, 100, 1000, and 5000 representing the number of additions performed by each activity.

Figures 3, 4, and 5 show the results of executing the three cases in both a sequential mode using one processor and in a parallel mode using 10 processors. In the parallel mode, a total of 12 processors was available. However, the main task and another support task used two processors so that 10 processors were available for simulation activities. The execution time shown in the figures represents the cumulative time for replicating the test simulation 10 times for each case.

Figure 3 shows the obvious case of one activity obtaining no benefit from parallel processing. This case illustrates the expected result that the parallel mode is less efficient when one activity is involved. Figure 4 shows that in the five activity case, parallel processing begins to show a computational profit. The profit is small at first, however the profit margin increases as the granularity increases. Figure 5 further illustrates this point. This case has a better mapping of activities to processors. As a result, the profit from parallel processing is greater than in the five activity case.

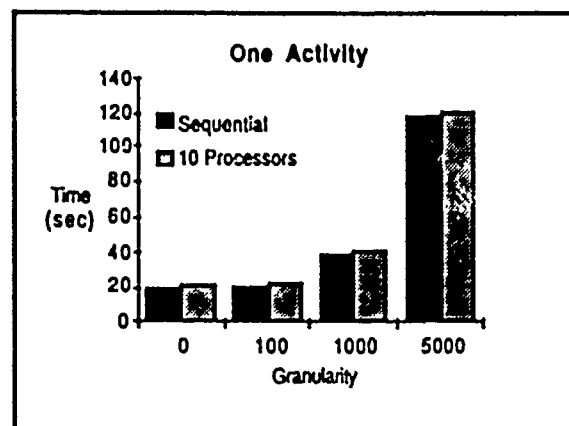


Figure 3 - Encore benchmark results for one activity

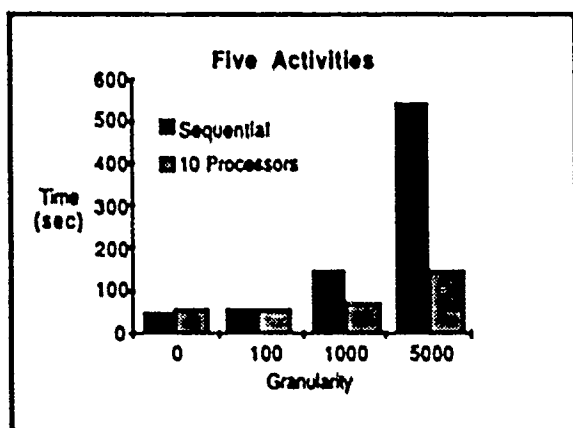


Figure 4 - Encore benchmark results for five activities

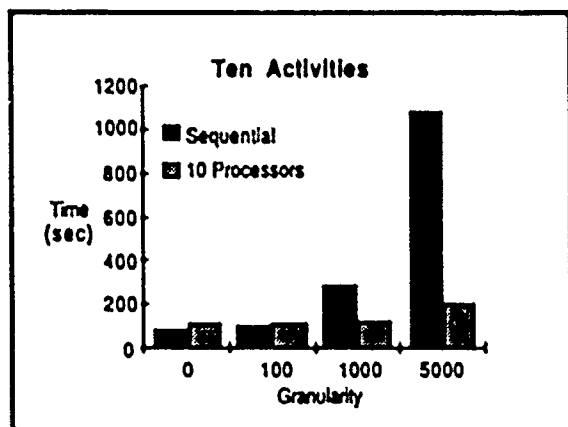


Figure 5 - Encore benchmark results for ten activities

Speedup is usually the bottom line when it comes to parallel processing benchmarks. In the cases performed, the maximum speedup achieved was 5.3 for ten processors and ten activities. However, had more cases been executed at higher granularities, the speedup would have approached 10.0, and a linear speedup could have been achieved. The point is that TASKIT is a set of tools, not an application, and speedup is an application dependent measurement.

It is difficult to extrapolate the results of the benchmark to make conclusions about the speedup that a particular simulation application might achieve by using TASKIT. There is no guarantee that a simulation using TASKIT will benefit from parallel processing. Some simulations may benefit while others may not. The test simulation represents an ideal case for an application that uses TASKIT. The results of the benchmark show that this ideal case performed as expected. It emphasizes the important role that

granularity plays in determining the benefit of parallel processing. Even the ideal case does not benefit from parallel processing if the amount of computation performed by the activities is not sufficient.

The Best of Both Worlds

As far as the model builder is concerned, TASKIT represents a low risk, yet a potentially high gain proposition. TASKIT is most beneficial for simulations that are well suited for parallel processing. Even if parallel hardware is not available, a simulation that is prepared for parallel processing can be developed and used in a sequential environment. When the appropriate parallel hardware becomes available, the simulation can be rehosted to it without modification.

For the case where a simulation is not inherently parallel, TASKIT is still a valuable tool. Since TASKIT is designed to execute efficiently in a sequential Ada environment, it can be used like any ordinary sequential simulation language. No additional language training is required for an Ada programmer to build simulations using TASKIT. The Ada programmer needs only to learn the TASKIT packages, not a specialized simulation language. The number of Ada programmers is growing as is the general level of experience with Ada. Additionally, the number of good Ada compilers is steadily increasing and Ada compilers are starting to emerge on personal computers. As a result, software and hardware support for TASKIT simulations is widely available.

The key to TASKIT's versatility is machine and environment independence. Virtually any computer that has an Ada compiler can be used for TASKIT simulations. TASKIT has been successfully tested for portability on a VAX using the VMS™ operating system, a Harris computer running under UNIX®, and the UNIX based Encore multi-processor used for the benchmark. TASKIT was successfully tested in these diverse environments without requiring any modification.

More importantly, TASKIT leaves the door open for future hardware advances. It is conceivable that someday multi-processors that support concurrent Ada could be available as desk top personal computers. With virtually no risk, simulation developers can begin preparing now for this future hardware by writing simulations in TASKIT and Ada today. Given the rate at which hardware technology is advancing and the typical lifespan of a simulation, it is a sound decision to design simulation software not only for the present, but also for the future.

VMS is a trademark of the Digital Equipment Corporation
UNIX is a registered trademark of AT&T

Other Services Provided by TASKIT

TASKIT is a robust simulation tool kit and provides other important simulation services in addition to activity management. These services include: statistics collection and reporting tools, a plotting package for displaying simulation statistics, a performance analyzer that collects data on the execution time of the simulation, resource protection tools to protect global data in a parallel environment, a random number and probability distribution package, keyed linked list services, and a simulation data management system that enables the analyst to input and manage simulation data in a spreadsheet format. The services that are provided are organized into Ada packages so that the model builder can selectively use only the services required.

Information on Obtaining TASKIT

TASKIT is part of the STARS Ada Foundations software repository and can be obtained from the Naval Research Laboratory. For more information contact:

Naval Research Laboratory
Code 5150, Bldg 1, Room 321
4555 Overlook Avenue, Southwest
Washington, D.C. 20375-5000

Biographies

MICHAEL ANGEL is a software engineer at General Dynamics-Data Systems Division. He has specialized in the design and development of simulations of space based systems. Mr. Angel holds a BS in physics/mathematics from the University of Illinois, Champaign Urbana. He is a member of the Society for Computer Simulation.

PAUL JUOZITIS is a software design specialist at General Dynamics-Data Systems Division where he has developed several large wargaming simulations for the Operations Research Department. His main interest is in the design and development of simulation software tools in Ada. Mr. Juozitis holds a BS in mathematics/computer science from the State University of New York at Binghamton and is a member of the Society for Computer Simulation.

Mailing Address:

General Dynamics
Data Systems Division
P.O. Box 85808
Mail Zone W2-5680
San Diego, CA 92138

Ada Run-Time Environment Considerations For Simulation

S. Shastry (201-758-7277)
Concurrent Computer Corporation
106 Apple Street, Tinton Falls, NJ 07724

Abstract

The choice of Ada as the implementation language for simulation systems raises some run-time environment concerns. In evaluating an Ada language system for simulation, these run-time environment issues must be considered. These issues include the support for migration from FORTRAN to Ada and the support for exploiting target system features.

Introduction

Over the years, some computer vendors have improved and tailored their machine architecture and the operating systems for simulation applications. We have seen systems evolve from a uni-processor configuration to multi-processor distributed system configurations. The concept of *shared memory* architecture was also introduced. Along with the evolution of the hardware configurations, the system software (specifically the operating systems and compilers) underwent changes to tailor machine-level access to the application requirements.

Traditionally, FORTRAN was used as the language for implementation of simulation software. Compiler vendors were free to enhance the language implementation to best suit their language offering to their architecture. Library routines were provided to exercise and control various system resources.

With the adoption of Ada as the language of implementation for most of the simulation software, some of the issues have to be approached in a slightly different fashion.

Compiler vendors do not have the freedom to enhance the language. They can only provide add-on features; however, these features are not supposed to change the semantics of the program as defined by the Ada language standard. The implementors of the simulation software are faced with using a new language system that seems to require new approach in the design. Therefore, implementors of both the simulation and the system software need to take a different approach to solve their problems.

Real-time application software (like a simulation system) has stringent timing constraints. It also has some special run-time requirements, in general. This paper addresses the following run-time environment issues from the point of view of implementing a real-time system in Ada:

- Migration from FORTRAN to Ada;
- OS tasking as opposed to Ada tasking;
- Access to operating system features;
- Shared memory configurations;
- Multi-processing system features; and
- Symbolic real-time debugging tools.

Migration from FORTRAN to Ada

In the past, the simulation software was mostly written in FORTRAN. It is possible that some parts of this software can be reused in new projects where Ada is being used as the implementation language. These parts can be rewritten in Ada in an incremental fashion, which provides a gradual migration path for the implementor. However, for this approach to work, the Ada language system should satisfy the following requirements which become significant if a large amount of FORTRAN code is reused. (The idea here is to use the FORTRAN code without modification.)

- (1) The Ada Compiler must support **pragma INTERFACE** to FORTRAN.

This pragma allows subprograms written in FORTRAN to be called from Ada procedures. The Ada language does not mandate Ada compilers to support this pragma; however, many Ada compilers do.

- (2) The interface between Ada and FORTRAN should be as *natural* as possible.

The source code for the call from Ada to FORTRAN should look as if the called procedure is written in Ada. No special requirements is to be put on the programmer in passing arguments to the called subprogram. For example, the Ada programmer should not be required to pass the address of the actual argument rather than the argument itself. This is mainly because the programmer should have to do minimum source modification when the FORTRAN subprogram is reimplemented in Ada. (The only source modification the programmer has to do is to remove the **pragma INTERFACE** for the subprogram and supply the body coded in Ada.)

Further, all relevant Ada data types must be mapped to the FORTRAN data types in a convenient fashion.

- (3) A mechanism to map Ada package specifications to FORTRAN COMMON blocks should be provided.

This is necessary for the simulation program that is made up of Ada and FORTRAN code to work on a single set of global data objects. In Ada, global objects are declared in package specifications and in FORTRAN, they are placed in COMMON blocks.

OS tasking as opposed to Ada tasking

The Ada language has constructs (known as **tasks**) to support parallel programming. These constructs are unsuitable in some applications where the task's priority has to be adjusted dynamically or the task is to be blocked/aborted if its time-frame has lapsed or a failure in the task needs to be notified to a monitor. Also, Ada tasking does not efficiently support periodic scheduling. Further, most of the implementations of Ada tasking are not efficient. There is a significant amount of overhead in a task switch. In a real-time environment, meeting the individual timing requirements of each task is important.

Further, a predictable behavior of the run-time system is crucial¹. Where these characteristics are less important, it is possible to adopt Ada tasking constructs. However, for more critical portions of the simulation software, it may be necessary to look for some other kind of support.

The operating system allows the user to define his own tasks (also known as processes). These processes can be individually controlled and monitored by another process called a monitor. The monitor can activate other subprocesses, adjust the system load, dynamically change the priority of processes, and so on. This gives the user some flexibility in the design of his simulation system. Major parts of the simulation system (e. g., Avionics, Visual, Engines, etc.) can thus be designed and implemented as separate processes with the main scheduler (or the operating system itself) controlling their execution. Large simulation programs may adopt this approach rather than using the Ada tasking model.

However, for the latter approach to work, the Ada system should allow the separate processes to share a common data. This data is normally declared in Ada package specifications. From the programmer's point of view, accessing objects in these package specifications should not require the knowledge of the tasking model. This also requires that there should be one and only one common data set loaded into the memory for the simulation system. Most importantly, there should be no run-time penalty for structuring the simulation system in this fashion. Further, the Ada language system should have facilities to enable a process written in Ada to control another process. The user should not have to write Assembly or FORTRAN code to achieve the desired functionality. These are examined in greater detail in the following sections.

Under the OS process model, one may require facilities for resource locking when there are more than one process in the application program. It may be necessary that when one process is manipulating a set of data, all other processes be denied access to it. It would be desirable to have resource locking facilities available at the Ada programming level. Therefore, the language system should provide procedures and functions to achieve data locking, for example, through Test-and-Set primitive operations.

Access to Operating System features

Most operating systems provide support for following facilities:

- *process manipulation and control* — This facilitates dynamic creation of processes, activating/blocking/aborting processes, dynamic adjustments of priorities, sensing state change of processes, and so on
- *process communication* — This allows interprocess communication for sending and receiving messages, data, notification of status, and so on
- *timer management* — This allows setting of periodic timer interrupts for regular system monitoring and load adjustments, and time-slice management
- *high speed input/output* — allowing for rapid data collection and transfer.

Efficient implementations of these functions is required for a simulation application program to function well.

These OS functions should be made available to the Ada programmer as procedure/function calls. Errors generated by these support routines must be mapped to exceptions so that the program can react to these errors in a consistent fashion. These OS functions should be in the standard library that is supplied by the compiler implementor. The implementor should supply package specifications defining the interface to these functions. The simulation programmer should not have to write these packages.

The Ada Real-Time Environment Working Group (ARTEWG) (under the guidance of the Ada Joint Program Office) is working on standardizing the run-time interface to some of the required OS features. When this standard is established and adopted by the compiler vendors, the Ada programmer will be able to make use of these features. However, he will have to evaluate the supplied features from the point of efficiency of its implementation.

Shared memory configurations

For a very large simulation, the software may require more than one processor for execution. One set of processes of the simulation software may run on one processor, second set on a second processor, and so on. However, there is a certain amount of data that

must be shared by some or all of these processes. Some hardware vendors have come up with loosely coupled systems configured in such a fashion that the various processors in the system share memory. That is, a certain part of the system memory is common to all of the processors in the configurations. The data of the simulation software that is common to the different processors is placed in this shared memory. In such a configuration, the only way of communication between processes running on the different processors may be through the shared memory.

Although there is a hardware solution to the problem, the software solution needs to be evaluated. The data common to more than one process running on different processors in this configuration needs to be placed in the shared memory. The simulation software designer needs to know this configuration merely for organizing his data structures. It should not have a major design impact for the designer. The programmer of such software should not have to know where the common data (which, again, is in an Ada package specification) is located at run-time. Further, there should not be any performance penalty for choosing the shared memory configuration. The compiler vendor should supply the tools required to build data images for these shared data, to build executable processes using the shared data, and to load the shared memory with the appropriate data.

Multi-processing system features

There is another hardware solution for some of the large simulation systems: a closely coupled multi-processing system. All of the processors in such a system share all of the available memory, and typically, one of the processors in the system performs the supervisory functions. In such systems, processes can communicate with processes running on the same system by sending messages or through shared data.

In a multi-processing system, the process scheduling may be done in one of three different ways: through programmer control, manual intervention, or operating system control. Most simulation programs prefer the first approach over the other two. This gives the simulation designer a closer control of the processes in the system. The functions that are necessary are: scheduling processes for execution on a processor in the multi-processor system, removing a process that is executing on a

processor when its time frame (or time slice) has expired, or moving a process from one processor queue to another. Once again, the compiler vendor should supply the multi-processing support functions and the associated package specification in the standard software package if the simulation software requires programmer (dynamic) control for process scheduling. In addition, the requirements listed in the section on shared memory configurations are also applicable.

Symbolic real-time debugging tools

The Ada Programming Support Environment (APSE) should provide tools for real-time debugging. These tools should not require the user to refer to machine locations in the program, but should use symbolic names from those programs. A symbolic real-time debugger is required. However, conventional debuggers are known to alter the real-time behavior of simulation programs.

When a program has reached the semi-production level, the user can use a symbolic real-time monitor². Such a tool will allow the user to monitor certain data objects in real-time without impacting the performance or the real-time nature of the program. This is due to the fact that the tool does not interrupt the execution of the program to provide information about the program being debugged. When the real-time monitor shows a potential incorrect execution in the program, the user should be able to activate a debugger and start debugging in the conventional fashion. These debugging tools should operate with the source level reference from the user.

The symbolic debugging tools should also have the ability to introduce faults into the program for testing or training purposes and enable checking of all possible logic paths in the program. Further, these debugging tools should be able to support the application program running in a multi-processing shared memory configuration.

Conclusion

Various run-time environment factors need to be looked at when evaluating an Ada Programming Support Environment for simulation. These factors are not just limited to the run-time speed of the code generated by the compiler or the efficiency of the core run-time system facilities. One needs to evaluate the language support for mixed language programming, support for shared memory

configurations, integration with the underlying operating system support, and real-time debugging facilities.

References

1. J. Stankovic, "A Serious Problem for Next-Generation Systems", *Computer*, Vol. 21, No. 10, October 1988, pp. 10-19.
2. S. Shastry, "Ada Language Symbolic Real-Time Monitor (SRTM)", *Tools for the Simulation Profession, 1988*, From the Proceedings of the 1988 Conference: Tools for the Simulationist Simulation Software, The Society for Computer Simulation International, pp. 1-4.



About the Author

S. Shastry is the Senior Manager of the Ada compiler development team at Concurrent Computer Corporation in Tinton Falls, New Jersey. He has held management and technical positions in the development of a universal optimizer for FORTRAN VII at Concurrent. He was also a member of a development team that implemented a fast memory-resident FORTRAN compiler at the Indian Institute of Technology, Bombay. He has 12 years of software development experience in compilers, run-time systems, and language tools.

Shastry received a MS (1980) in computer science from the University of Pennsylvania, Philadelphia, MTech (1976) in computer science from the Indian Institute of Technology, Bombay, India, and a BE (1974) in electrical communications from the Indian Institute of Science, Bangalore, India.

BENCHMARKING THE REAL-TIME PERFORMANCE OF DYNAMIC ADA PROCESSES

Alice J. Lee William R. Maere Donna K. Dot

Lockheed Electronics Co., Inc.
Route 22, Plainfield, NJ 07061

ABSTRACT

In designing the multi-tasking signal processing software, it is important to know the dynamic behavior and timing performance of the process model in a real-time environment. However, due to the hardware and software restrictions of an embedded system, the designer may not have all the information necessary to fully understand the model or the convenience to optimize the model's performance. This paper describes restrictions encountered and results obtained in benchmarking the real-time processes, originally written in native assembly language, against the same processes coded in Ada. Both executable codes were downloaded into a Motorola 68010 embedded microprocessor to obtain the benchmarks. Simulation of the same processes in Ada and executing them on a self-targeted VAX/8600 demonstrated that the model characteristics can be thoroughly studied and correlated to the benchmarks of individual procedures and tasks. Further, the task interactions, task overhead and the effect on the throughput will be analyzed by comparing a sequential model to a parallel tasking model. For time critical multi-tasking processes this study suggests that a rapid model simulation in Ada on VAX provides quick identifications of time critical regions, an understanding of model behaviors and allows the designer to tailor the model for the best timing performance prior to target implementation.

1. INTRODUCTION

The processes studied (Figure 1) involve the sorting of radar pulses that have been digitized and sampled to produce a 24-byte feature vector. The algorithm consists of a closed loop decision process that compares each of the 24 features to known libraries of feature vector limits in search of a "best fit". The best fit is identified in terms of a feature slot identification number (I.D.). Cluster analysis is used to identify signals that do not correlate with existing libraries either due to new signals introduced into the system or old signals whose features may be drifting because of environmental changes. In order to keep track of the continuous processes of clustering and producing statistically generated mean feature vectors, the processes were written in native MC68010 assembly language (ASM). As a result, the ASM working with custom built hardware is able to meet the real time requirement when running in an embedded 68010 target (TGT). The 24 features obtained via cluster analysis are used to set new WAM (window addressable memory) limits or to update old WAM limits. Each WAM slot provides 24 feature limits to allow pattern matching with the present signals. The whole processes cycle until all WAM slots are set. At least two tasks are needed to handle the asynchronous processes - an NS task to handle new signals and an AS task to adapt to drifting signals. Both tasks rely on input signal thus requiring two more tasks - a DMA task to input feature vectors through direct memory access and a SYNCHRONIZATION task to synchronize all three tasks while causing DMA to stay one buffer ahead of NS and AS. This arrangement allows the overlap of data I/O with CPU execution and ensures that buffered data does not become outdated.

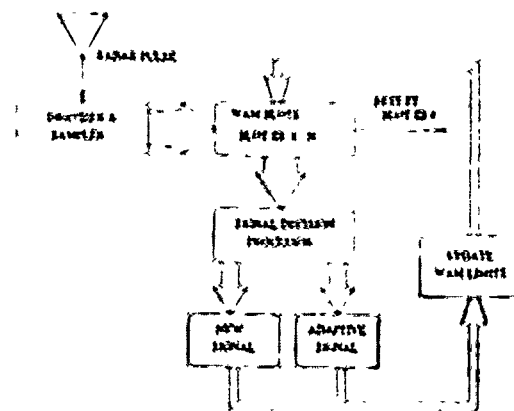


Figure 1. Functional Block Diagram of the Processes

In addition to tasks, a doubly linked list is used as the basic data structure by the process model. This structure suits the dynamic nature of the signals while providing pointers for direct data access. Whenever NS or AS accumulates enough links from the input buffer, signals on the linked list are sorted into a hash table to speed up clustering. Then signal links with small feature differences start to cluster by joining features of sub-clusters. To avoid waiting time in garbage collecting the used links, the model allocates sufficient free links from the heap then controls the link allocation and deallocation. In order to share the global link pool between NS and AS tasks, a FREE_POOL task has to be added to guarantee that the pool access is mutually exclusive. Figure 2 depicts the relationship of the five tasks in the model. To implement tasks, ASM was written around a popular "off-the-shelf" real-time operating system (O.S.). Ada, on the other hand, has concurrent programming facilities which provide the notational convenience and conceptual elegance in writing concurrent algorithms inherent in this model. Figure 3 depicts the Ada implementation of the model in five packages, the dependency of the packages and the entry calls made by the parallel tasks.

In the process to determine how close Ada meets the real time requirement, we encountered the following difficulties when benchmarking both Ada and ASM on TGT:

- (1) ASM benchmark - benchmarks of procedures directly related to I/O are not measured since I/O is a machine dependent variable. Meanwhile, timing of the task overhead can not be done without changing the source code. Therefore less than half of the procedures are benchmarked and no throughputs are measured. Moreover, the dynamic behavior of the procedure is not always known because

121 Ada benchmark - TSADATE68 cross compiler is found to be able to handle three tasks but not all five tasks needed by the processes (this problem was fixed in a later version 3.1.5). Despite various tries, execution of the five-task model always gets aborted once entering the multi-tasking stage. Without attempting other compilers, we analyzed the parallel model and determined that it can be sequentialized by modifying the top most hierarchy of the entire structure. Other than the procedures that replace the tasks, all remaining procedures stay the same for the two models. Therefore benchmarks of the common procedures should be the same for both models.

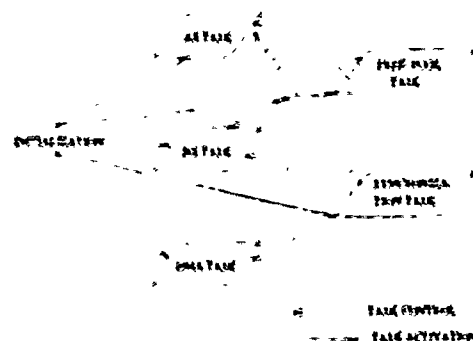


Figure 4 describes the major differences between the parallel model and the sequential model. In the sequential model, the MAIN program is used to scan the input buffer. If the signal scanned is new then MAIN will call NS for linking else it will call AS for linking. After all buffer signals are scanned, MAIN will call DMA to fill the next buffer address. On the contrary, in the parallel model both NS and AS scan the same buffer independently while DMA runs simultaneously. Although both models take the same input and generate the same output, their timing performance may differ significantly.

Although benchmarks of ASM and Ada can be done in TGT, they both encountered restrictions. To get around the problem we benchmarked the processes on a self-targeted machine, VAX/3600 (VAX), where the TSADA/VMS compiler is able to handle both models properly. When coding Ada, instead of replicating ASM step by step, we use Ada features to implement the equivalent ASM functions. Processes run on VAX will not be able to test the hardware I/O used on TGT, but will provide the designer the convenience to time and analyze high level algorithms of the model, and to have full control of the input data. Input data can be synthesized by a simulation program or converted from binary data collected from antennas. When benchmarking procedures of parallel processes, the time slicing feature is disabled in order to prevent task preemption or swamping from interfering.

The study is summarized in the following sections. System setup and timers such as logic analyzer, system clock (SYS clk), Ada clock (Ada clk) and target clock (TGT clk) are presented in section 2 of the paper. Section 3 discusses timer characteristics, model behavior, benchmark results and model comparisons, while Section 4 presents our conclusions.

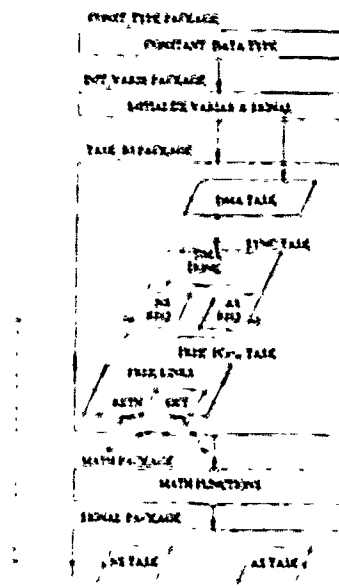


Figure 3. Ada Package Construct of the Parallel Model

2 SYSTEM SETUP AND BENCHMARK TOOLS

2.1 System Setup

Since the benchmark measures both machine architecture and the ability of the compiler to generate efficient code[1,4], major system configurations of VAX/3600 and TGT/68010 are listed in Table 1.

2.2 Benchmark Tools

2.2.1 ASM on Embedded TGT

The 68010 code was assembled on a Kontron Development System then downloaded onto the TGT operated under an UNIX like O.S. - PSOS. When benchmarking, the HP-1650A logic analyzer, HP-10311B interface probe and HP-10269C preprocessor were used to provide precise timings between trigger points.

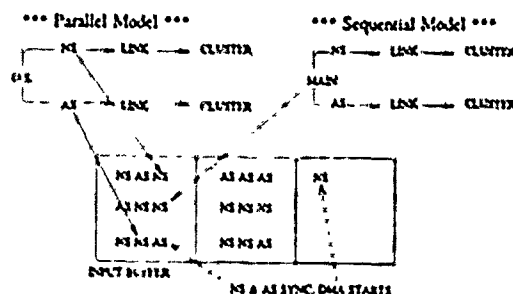


Figure 4. Parallel Model vs Sequential Model

2.2.2_Ada on Embedded TGT

The Ada execution environment has to be retargeted and adapted to the machine dependent portions of the TGT including console I/O, clock control, interrupt handler and exception table. To allow benchmarking, a routine which controls a programmable countdown timer (PTM MC68840) was added to the Ada environment module. Then the Ada code was cross compiled on VAX by TSADA/E68 and the resulting S-recode was downloaded onto TGT. Since the Ada executive kernel has been linked into the download module, the TGT is treated as a bare machine with no need of the original O.S.

2.2.3_Ada on VAX

The Ada code was compiled via TSADA/VMS compiler on VAX then benchmarked on VAX using SYS clk. SYS clk called from a Fortran routine provides better measurements than Ada clk called from the Calendar package although portability of SYS clk is restricted to machines of VAX series. Unlike the logic analyzer, neither TGT clk nor SYS clk provided timing accuracy at the nanosecond level thus a timing loop technique was used to improve the accuracy of both clocks. The benchmark of a procedure was obtained by subtracting the benchmark of a control loop from that of a test loop[2].

For all Ada versions, signal data were initialized in memory in order to minimize the I/O time spent in throughputs. Meanwhile no optimization was invoked to ensure the proper benchmarking[2].

3. RESULTS

3.1 Timer Characteristics on TGT and VAX

Among the three clocks of Ada clk, SYS clk and TGT clk, Table 2 shows that TGT clk has the highest resolution, the lowest overhead and the least interferences.

Due to the high resolution of SYS clk and the slowness of the cross compiled Ada, the benchmarks of Ada on TGT were obtained without looping in most cases. To benchmark on the VAX, we choose SYS clk because it offers a 20% better resolution than Ada clk and SYS clk measures CPU time only. Notice that the CPU time of a process is still affected by page thrashing or swapping from the O.S.[2,3]. However, excessive page thrashing is unlikely to occur under a single user condition during benchmarking although some will occur to allow the O.S. to perform minimum system services. Benchmarks on the dedicated TGT, however, are not interfered by the system at any time. System interferences, reflected by the standard deviation (STDEV) of the measurement mean, is near 0% for TGT clk and less than 2% for SYS clk. Interferences of VAX system work load on both Ada clk and SYS clk are displayed in Figure 5.

3.2 Benchmark Results on TGT and VAX

Table 3 shows the benchmark comparisons of parallel Ada vs sequential Ada on VAX and sequential Ada vs parallel ASM on TGT. Benchmarks ranging from 7 usec to 1,240,000 usec are specified for procedures or tasks that are indented according to their hierarchical calling sequence. To avoid repeating, we elaborate the sequence only when the procedure is first encountered. In comparing the two Ada models on VAX, benchmarks are listed according to procedures that are common to both models and unique to each model. Among the benchmark differences, the parallel model exhibits a much slower throughput than the sequential model because of the task overhead. Meanwhile, the ASM benchmarks, though limited, indicate that ASM runs roughly 10 times faster than the cross compiled Ada on TGT. Therefore we conclude that, if the parallel model used by ASM were coded in Ada, the efficiency of the Ada parallel model will be more than 10 times slower than ASM.

PARAMETERS	VAX	TGT
OPERATING SYSTEM	VMS 4.7	KERNEL/PSOS
COMPILER	TSADA/VMS 3.13	TSADA/E68 3.13
MODEL	6600	IRONICS IV1601
CPU	DIGITAL	68010
CLOCK RATE	12.5 MHz	12.5 MHz
CACHE MEMORY	1K BYTE	NO
MAIN MEMORY	64 KBYTE	512 KBYTE
FP COPROCESSOR	NO	NO
FP ACCELERATOR	YES	NO
DATA BUS	32 BITS	16 BITS
INTERNAL REGISTER	32 BITS	32 BITS
MEMORY MANAGEMENT UNIT	YES	NO

TABLE 1. SYSTEM CONFIGURATIONS

PARAMETER	ADA CLK	SYS CLK	TGT CLK
MEASURING TIME	ELAPSED	CPU	CPU
INTERFERENCES	O.S. & OTHERS	O.S.	NONE
RESOLUTION, USEC	10000	10000	6.4
OVERHEAD, USEC	141	117	6.4
SPREADING/10MSEC	71	85	1362

TABLE 2. TIMER CHARACTERISTICS

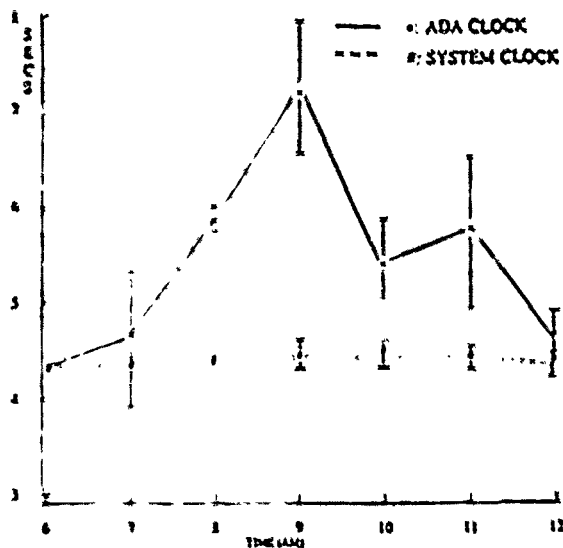


Figure 5 Effect of VAX System Work Load on Ada clk & SYS clk

3.3 Long-Integer vs Fixed-point

In ASM, the computation part of CALC_WAM_LIMITS procedure is operated via fixed point math, mainly by shifting bits. In order to preserve the accuracy while calculating the means and standard deviations of the 24 features. To code the same function in Ada both the long-integer method and the fixed-point method are feasible. The long-integer method preserves the accuracy by simulating bit-shifting via multiplication while the fixed-point method does it automatically via type-casting. Although the fixed-point method is easier to implement, its execution time of 982 usec is 24% longer the 794 usec of the long-integer method on VAX. Thus we choose the long-integer method.

PROCEDURES/TASKS IN HIERARCHICAL ORDER CODE/MACHINE PROCESS MODEL	BENCHMARKS IN USEC				
	ADA/VAX			ADA/TGT ASM/TGT	
	PARA*	COMMON	SEQ**	SEQ**	PARA*
0. INITIALIZATION		34003		135859	++
1. DMA_TRANSFER PROCEDURE/TASK (1 INPUT BUFFER, WAM#-11) ACCESS 1 PULSE FROM MEMORY	40190		41337	409383	++
		7		97	++
2. NS PROCEDURE/TASK (AVG)	101923		50692	607802	++
LINK_LOGICAL_PHYSICAL_REC	11969		286	1703	++
RETN_LOGICAL_REC	1209		44	182	++
RETN_CLUSTER_INX		9		45	++
DEALLOC_USED_REC	--		11	32	--
FREE_POOL.DEALLOC_USED_REC	1177		--	--	++
ALLOC_NEW_REC	--		17	70	--
GET_LOGICAL_REC	1199		--	--	++
FREE_POOL.ALLOC_NEW_REC	1190		--	--	++
GET_PHYSICAL_DATA (MINIMUM)	15		--	--	++
CLUSTER_DISTANCE (AVG) (WITH 13 LINKS)*		15073		135985	++
RESET_SORTING_TBL		411		176401	8570
DIFFERENCE		197		2822	334
INSERT_SORTING_TBL		18		1863	120
CLUSTER_LINKING		10		70	24
CLUSTER_RUNNING_MEAN		6153		75275	12504
RETN_CLUSTER_INX		312		6952	863
GET_CLUSTER_INX		9		45	11
NS_CLUSTER (WAM#-11)		11		45	13
(CF 1ST CLUSTER)*		4249		58291	++
IS_NEW_CLUSTER (WAM#-11)		2400		24013	2112
CALC_WAM_LIMITS		1127		34563	1941
UPDATE_WAM_LIMITS		197		21480	1336
RETN_LOGICAL_REC	1209		44	3097	59
				182	67
3. AS PROCEDURE/TASK (AVG)	172500		125500	1238067	++
LINK_LOGICAL_PHYSICAL_REC	38818		850	5310	++
ALLOC_NEW_REC	--		17	70	--
GET_LOGICAL_REC	1199		--	--	++
GET_PHYSICAL_DATA (MINIMUM)	15		--	--	++
CLUSTER_DISTANCE (AVG)		99158		925010	++
DIFFERENCE		197		1863	120
AS_CLUSTER		9185		167559	++
CALC_WAM_LIMITS		1127		21480	1336
RETN_LOGICAL_REC	1454		228	1666	++
DEALLOC_USED_REC	--		11	32	--
FREE_POOL.DEALLOC_USED_REC	1177		--	--	++
AS_PURGE (MINIMUM)	232		--	--	++
4.1. SET_NXT_BUF_INX	--		84	742	--
4.2. SYNCHRONIZATION TASK	1247		--	--	++
ACCEPT_NS_REQUEST					
ACCEPT_AS_REQUEST					
ACCEPT_DMA_DONE					
SET_NXT_BUF_INX	84		--	--	++
5. FREE_POOL TASK					
ACCEPT_ALLOC_NEW_REC	1190		--	--	++
ACCEPT_DEALLOC_USED_REC	1177		--	--	++

* PARALLEL

** SEQUENTIAL

-- NOT APPLICABLE

++ NOT MEASURED

* MEASURED AT CONDITIONS SPECIFIED TO THE LEFT

TABLE 3. BENCHMARKS OF ADA ON VAX & TGT AND ASM ON TGT

3.4 Tasking Overhead

The tasking feature of Ada, though powerful in providing concurrent solutions, is severely restricted by its run time performance. While a procedure call invokes the save and restore of the return address and registers used, a task entry call invokes the more extensive context switch. During the switch, the old task is preempted and its context layer saved then a new task is selected and its context layer restored. Since a context layer is the task execution environment, it includes sets of registers and all local variables used by the task. On both TGT and VAX, Table 4 demonstrates that task entries called in parallel model consumes 50-100 fold more time than procedures called in sequential model. Moreover, the actual work conducted within the `ALLOC_NEW_REC` and the `DEALLOC_USED_REC` entries of the `FREE_POOL` task is minor. Nevertheless, the two entries are needed in order to guarantee the mutual exclusive access of the global free pool between NS and AS tasks. If NS and AS keep their own free pool, the `FREE_POOL` task can be eliminated toward the benefits of the process efficiency.

		PARALLEL	SEQ**	RATIO
	# OF TASKS	ENTRY, USEC	ENTRY, USEC	CALL, ENTRY/ CALL
VAX	SYNCHRONIZATION	3 1247	9.5%	0 --
	ALLOC_NEW_REC	2 1190	1.5%	17 70
	DEALLOC_USED_REC	2 1177	2.7%	11 107
TGT	SYNCHRONIZATION	3 2040	0%	0 --
	ALLOC_NEW_REC	2 3380	0%	70 48
	DEALLOC_USED_REC	2 3380	0%	32 106

* TGT BENCHMARKS ARE OBTAINED FROM SIMULATION WITH NULL STATEMENT INSIDE ENTRY ** SEQUENTIAL

TABLE 4. TASK OVERHEAD - ENTRY CALL VS PROCEDURE CALL

3.5 Throughput and Suppression

The average time needed to update the 24 features of 1 WAM slot, defined as throughput, by NS and AS tasks are compared. Table 5 indicates that the sequential Ada model runs 10 times slower on TGT than on VAX. Meanwhile, on VAX, the task overhead associated with the parallel model makes it run 50% slower than the sequential model, and the percentage of being slow increases after suppressing all run time checks. By calling `FREE_POOL` task less often, AS invokes less task overhead than NS. On the other hand, AS consumes 50-100% more time in throughput than NS even when AS uses simpler algorithm. Benchmarks of `CLUSTER_DISTANCE` procedures from NS and AS indicate that this is because AS collects twice as many signals and the time spent in pairing signals for feature difference increases by the power of two.

CPU	MODEL	NS USEC	NS RATIO	AS USEC	AS RATIO	AS/NS RATIO
UNSUPPRESSED VERSION						
VAX	PARA*	101923	1.60	172500	1.37	1.69
VAX	SEQ**	60692	1.00	125500	1.00	2.07
TGT	SEQ	607803	10.01	1238067	9.87	2.04
SUPPRESSED VERSION						
VAX	PARA	90130	1.91	135000	1.41	1.50
VAX	SEQ	47231	1.00	95000	1.00	2.01
TGT	SEQ	561488	11.89	1123043	11.82	2.00

* PARALLEL ** SEQUENTIAL

TABLE 5. NS & AS THROUGHPUT - SUPPRESS VS UNSUPPRESS

For better efficiency, all Ada run time checks are suppressed and the results are displayed in Table 6. The suppressed versions are able to decrease the memory size of the executable code by 5%, and to increase the throughput by 10-25% depending on the model and the machine on which it runs.

PROCESS	CPU	MODEL	MEMORY	TIME
NS	VAX	PARA	4.1%	11.6%
AS	VAX	PARA	5.2%	21.7%
NS	VAX	SEQ	1.0%	22.2%
AS	VAX	SEQ	4.9%	24.3%
NS	TGT	SEQ	5.4%	7.6%
AS	TGT	SEQ	5.4%	9.3%

TABLE 6. MEMORY AND TIME SAVED BY SUPPRESSION

3.6 Benchmark Equation Derivation

Since both models are dynamically driven by I/O, benchmarks of I/O dependent procedures appear to be random at first glance. However, by matching benchmarks with activities traced from a program independent of the benchmark study, we are able to relate the benchmarks to events that occur dynamically within the procedure and to formulate the benchmark into an event equation:

$$\text{benchmark} = \text{deviation} + \text{sum of all} \\ (\text{sub-procedure benchmark} * \# \text{ of calls}).$$

where deviation reflects the measurement error (<1%, Table 8) or procedure overhead (Tables 7,9,10). Table 7 shows a simple example of procedure `Cluster_Distance` of AS task making 496 calls of `DIFF` sub-procedure. The difference between the measured benchmark 925010 usec and the calculated `DIFF` benchmark total, $496 * 1863$ usec, results in a deviation of 927 usec on TGT. Similarly the deviation is calculated to be 1694 usec on VAX. Tables 8 and 10 describe that more complicate equations can be formed for procedures whose individual benchmarks vary with the number of sub-procedures called during each measurement. Table 9 shows a procedure, though makes no sub-procedure calls, whose benchmark is directly related to the number of WAM set. Equations thus derived allow run-times to be predicted under various conditions without actual benchmarking.

3.6.1 CLUSTER_DISTANCE called by AS

ADA/TGT USEC	#OF DIFF CALLS	DEVIATION USEC	ADA/VAX USEC	DEVIATION USEC
925009.9	496	927.2	99158.3	1694.3

$S = [N * (N-1) / 2]$ WITH $N = \# \text{ OF LOGICAL LINKS} = 32$

TABLE 7. CLUSTER_DISTANCE CALLED BY AS

$$\begin{aligned} \text{CLUSTER_DISTANCE, USEC} &= \text{DEVIATION} + \\ &(\# \text{ OF DIFF CALLS}) * (\text{DIFF BENCHMARK}) \\ &= 927.2 + [N * (N-1) / 2] * 1863.1 \quad \text{-- ADA/TGT} \\ &= 1694.3 + [N * (N-1) / 2] * 196.5 \quad \text{-- ADA/VAX} \end{aligned}$$

Because of the high calling frequency of `DIFF` by `CLUSTER_DISTANCE` procedure, `DIFF` consumes 99% of the `CLUSTER_DISTANCE` time on both VAX and TGT. If `DIFF` were coded in-line, `DIFF` will become 19% more efficient on VAX but only 1% more on TGT.

3.6.2 CLUSTER_DISTANCE called by NS

ADA/TGT USEC	RESET*	DIFF*	INSERT*	USEC	ADA/VAX USEC	DEVIATION USEC
89967.4	1	45	45	138.8	10230	173.4
151144.3	1	78	42	45.7	14435	-50.7
109260.2	1	55	55	96.9	12330	132.4
107633.9	1	55	31	160.3	11880	109.6
129065.0	1	66	46	41.5	14250	51.1
151980.2	1	78	56	-104.1	16660	-74.9
128377.6	1	66	30	480.6	13950	35.9
153509.1	1	78	78	-123.9	17080	-46.5
151475.8	1	78	48	-45.2	16550	-42.5
175865.6	1	91	51	-86.6	19040	-160.4
152067.2	1	78	56	-17.1	16685	-49.9
150825.0	1	78	38	7.9	16335	-79.5
152016.6	1	78	56	-67.6	16705	-29.9
109248.6	1	55	55	85.4	12360	164.2
128013.4	1	66	30	116.4	13930	15.9
153538.6	1	78	78	-94.5	17095	-31.5
89991.0	1	45	45	162.5	10255	200.4
128333.4	1	66	34	154.8	14040	54.7
176935.0	1	91	67	-143.5	19310	-175.2
150800.0	1	78	38	-17.1	16350	-64.5
MEAN =				39.6		6.6
STDEV =				145.1		106.8
RESET* =				RESET_SORTING_TBL	DIFF* = DIFFERENCE	
INSERT* =				INSERT_SORTING_TBL		

TABLE 8. CLUSTER_DISTANCE CALLED BY NS

$$\begin{aligned}
 \text{CLUSTER_DISTANCE, USEC} &= \text{DEVIATION} + (\text{RESET BENCHMARK}) + \\
 & \quad (\# \text{ OF DIFF CALLS}) * (\text{DIFF BENCHMARK}) + \\
 & \quad (\# \text{ OF INSERT CALLS}) * (\text{INSERT BENCHMARK}) \\
 &= 39.6 + 2822.4 + (\# \text{ OF DIFF CALLS}) * 1063.1 + \\
 & \quad (\# \text{ OF INSERT CALLS}) * 70.4 -- \text{ADA/TGT} \\
 &= 6.6 + 411.1 + (\# \text{ OF DIFF CALLS}) * 196.5 + \\
 & \quad (\# \text{ OF INSERT CALLS}) * 17.8 -- \text{ADA/VAX}
 \end{aligned}$$

3.6.3 IS_NEW_CLUSTER called by NS

WAM# SET	ADA/TGT USEC	WAM#	ADA/VAX USEC	WAM#	ASM/TGT USEC	WAM#
0	42.9	--	10.7	--	21.1	--
1	3182.1	3139.2	226.7	216.0	188.3	167.2
2	6317.4	3137.3	446.0	217.7	359.3	169.1
3	9456.6	3137.9	663.3	217.6	542.0	173.6
4	12595.8	3138.2	881.3	217.7		
5	15731.2	3137.7	1096.0	217.1		
6	18867.8	3137.5	1312.0	216.9		
7	22009.6	3138.1	1532.0	217.3		
8	25143.0	3137.5	1746.7	217.0		
9	28284.2	3137.9	1967.3	217.4		
10	31420.8	3137.8	2186.7	217.6		
11	34563.2	3138.2	2400.0	217.2		
12	37692.2	3137.4	2618.0	217.3		
MEAN =		3137.9		217.2		170.0
STDEV =		0.5		0.5		3.3
* = (BENCHMARK AT WAM#N - BENCHMARK AT WAM#0) / #N						

TABLE 9. IS_NEW_CLUSTER CALLED BY NS

$$\begin{aligned}
 \text{NS_CLUSTER, USEC} &= \text{DEVIATION} + \\
 & \quad (\# \text{ OF WAM SET}) * (\text{WAM BENCHMARK}) \\
 &= 42.9 + 3137.9 * (\# \text{ OF WAM SET}) -- \text{ADA/TGT} \\
 &= 10.7 + 217.2 * (\# \text{ OF WAM SET}) -- \text{ADA/VAX} \\
 &= 21.1 + 170.0 * (\# \text{ OF WAM SET}) -- \text{ASM/TGT}
 \end{aligned}$$

3.6.4 NS_CLUSTER called by NS

ADA/VAX USEC	RETH*	LIMITS*	# OF WAM SET	DEVIATION USEC
452.7	7	0	0	131.8
1899.3	12	1	0	229.8
607.3	10	0	0	153.6
1942.0	9	1	1	188.2
564.7	9	0	0	155.2
2300.7	11	1	2	241.0
2407.3	9	1	3	219.1
620.7	10	0	0	166.9
2622.0	9	1	4	216.5
674.7	11	0	0	176.6
2877.3	10	1	5	210.3
3106.0	10	1	6	221.7
3428.0	12	1	7	237.9
550.7	9	0	0	141.2
3481.3	9	1	8	207.0
3903.3	13	1	9	234.5
501.3	8	0	0	136.2
517.3	8	0	0	152.2
4032.0	11	1	10	234.6
4248.7	11	1	11	234.0
MEAN =				194.0
STDEV =				38.8

RETH* = RETH LOGICAL REC
LIMITS* = CALC WAM LIMITS
WAM* = # OF WAM SET IN IS_NEW_CLUSTER CALL

TABLE 10. NS_CLUSTER CALLED BY NS

$$\begin{aligned}
 \text{NS_CLUSTER, USEC} &= \text{DEVIATION} + \\
 & \quad (\# \text{ OF RETH CALLS}) * (\text{RETH BENCHMARK}) + \\
 & \quad (\# \text{ OF LIMITS CALLS}) * (\text{LIMITS BENCHMARK}) + \\
 & \quad (\text{IS_NEW_CLUSTER BENCHMARK}) \\
 &= 194.0 + (\# \text{ OF RETH CALLS}) * 44.3 + \\
 & \quad (\# \text{ OF LIMITS CALLS}) * 1127.2 + \\
 & \quad 10.7 + (\# \text{ OF WAM SET}) * 217.2 -- \text{ADA/VAX}
 \end{aligned}$$

3.7 Empirical Factors Correlating Benchmarks

Since VAX benchmarks are readily available, we suspect that TGT benchmarks may be correlated to VAX benchmarks via empirical factors. When a benchmark comparison is made on different processors or different systems, the empirical correlating factors become a gross measurement of language influence, compiler influence, machine architecture, cache influence, etc.[1,4]. These factors, however rough, allow the designer to guess the approximate timing performance of the model and possibly tune the model before embarking on a full scale benchmark study on TGT. Since only the sequential Ada model runs on TGT, we tested its benchmark correlation to the same model run on VAX. Table 11 lists the resulting factors correlating Ada procedures of similarly features. The correlation is then tried between benchmarks of Ada and ASM, both run on TGT, and finally between Ada on VAX and ASM on TGT. Results of the two sets of correlations are shown in Table 12. To the thirteen benchmarked ASM procedures, only those that are not hand optimized and are using the same data structure and algorithm as the Ada code can be used. Based on the six qualified procedures, the factor correlating Ada on VAX and ASM on TGT is estimated to be 1.0(+/-0.25). The designer, however, is cautioned to use the factor only for codes intended for similar purpose and developed under the same systems.

MAIN FEATURE	PROCEDURES	FACTOR, TGT/VAX MEAN(STDEV)
GENERAL	DMA, NS & AS THROUGHPUT, NS & AS CLUSTER DISTANCE, DIFFERENCE, SET_NXT_BUF_INX	9.5(0.4)
POINTERS	NS & AS LINK LOGICAL PHYSICAL, REC, RETN LOGICAL LINK, GET_ LOGICAL_REC, INSERT_SORTING_TBL, RESET_SORTING_TBL	5.2(1.3)
LONG INT- GER MATH	CALC_WAN_LIMITS, AS_CLUSTER	18.7(0.6)
MEMORY ACCESS	ACCESS 1 PULSE FROM MEMORY, IS_ NEW_CLUSTER, UPDATE_WAN_LIMITS	14.5(1.1)

TABLE 11. CORRELATE BENCHMARKS OF ADA/VAX TO ADA/TGT

MAJOR FEATURE	PROCEDURES	TGT/VAX FACTOR, MEAN(STDEV)	ASM/VAX
ASSIGNMENTS & POINTERS	RETN_CLUSTER_INX, GET_CLUSTER_INX, RESET_SORTING_TBL, INSERT_SORTING_TBL	5.0(1.3)	1.2(0.2)
MEMORY ACCESS & SIMPLE MATH +,-	DIFFERENCE, IS_NEW_CLUSTER	13.3(1.6)	0.8(0.1)
COMBINED RATIO		1.0(0.25)	

TABLE 12. CORRELATE BENCHMARKS OF
ADA/TGT TO ADA/VAX AND ASM/TGT TO ADA/VAX

3.8 Module Sizes and Ease of Modeling

Table 13 compares the code sizes of Ada and ASM both run on TGT. For a fair comparison the Ada code is compiled under conditions similar to ASM with no run time checks, no TEXT_IO and no pulse initialization. Further, the DMA module which simulates the hardware function used by ASM is excluded from the Ada code.

PARAMETER	ADA CODE	ASM CODE
KERNEL/O.S. SIZE	31 KBYTE	5 KBYTE
EXE CODE SIZE	20 KBYTE	5 KBYTE
DATA SIZE	192 KBYTE	8 KBYTE
SOURCE CODE	870 LINES	2360 LINES
* COMMENT LINES EXCLUDED		

TABLE 13. MODULE SIZES OF ADA VS ASM ON TGT

As expected, ASM modules are smaller than Ada modules except the amount of the source code. By using the bit fields, the ASM is able to have a data size much smaller than that of Ada. If Ada were to use representation clauses, the data size will decrease but code size will increase in order to unpack the fields for accessing. Meanwhile, the readability and versatility of Ada makes the modeling concept easy to realize. For example, the parallel model containing five tasks is implemented with 10% increase or 15% total change on the source code of the sequential model and with all changes localized in two of the five packages.

4. CONCLUSION

In spite of limitations encountered in benchmarking parallel processes on TGT, we are able to demonstrate that using Ada on VAX allows rapid model simulation of real time processes and quick profiling of the process performance. Without I/O restrictions and lengthy downloading time, this approach encourages the designer to try different model constructs and language features in search of a model with desired characteristics and required timing performance. Further, the study shows that benchmarks of dynamic processes can be formulated into event equations to allow benchmark prediction. It also suggests that an empirical factor may relate the VAX benchmark to the TGT benchmark although more work needs to be done to confirm such a relationship. Finally, on the embedded TGT, the real time performance of the cross compiled Ada is not yet close to that of ASM which strives to take advantage of the underlying hardware and language constructs for efficiency.

5. ACKNOWLEDGEMENT

The authors would like to express their appreciation to Mike Cahill for supporting the hardware and providing the test algorithms and to David Ihsu for reviewing this article.

6. REFERENCES

- [1] Reinhold P. Weicker, "Drystone: A Synthetic Systems Programming Benchmark", Communications of the ACM 27(10), 1013-1030, October, 1984.
- [2] Russell M. Clapp et al., "Toward Real-Time Performance Benchmarks for Ada", Communications of the ACM 29(8), 760-778, August, 1986.
- [3] Neal Altman, "Factors Causing Unexpected Variations in Ada Benchmarks", Technical Report, CMU/SEI-87-TR-22, ESD-TR-87-173, October, 1987, Carnegie-Mellon University.
- [4] Jack Dongarra et al., "Computer Benchmarking: Paths and Pitfalls", IEEE Spectrum, 38-43, July, 1987.

7. AUTHORS

Alice J. Lee is a member of Information Systems group at Lockheed Electronics. She has implemented several real-time systems including direction finding, gun fire control and signal processing. She received an M.S. degree in Computer Science from New Jersey Institute of Technology.

William R. Macre is a Software Staff Engineer at Lockheed Electronics in Plainfield, N.J.. He has been involved the design and implementation of several real-time embedded systems targeting microprocessors. This involvement has lead to his interest in real-time Ada and Ada related topics. He received a B.S.E.E. degree from New Jersey Institute of Technology.

Donna K. Doi is the Program Manager for Advanced Programs in the Information Systems line of business group. She has participated in the development of real time systems and Ada software development. She has a Master's degree in Engineering from California State University, Fullerton, and a Master's degree in Mathematics from the University of Arizona.

ESTABLISH AND EVALUATE ADA RUNTIME FEATURES OF INTEREST FOR REAL-TIME SYSTEMS

Sharon Lofkowitz
IIT Research Inst.
4600 Forbes Blvd.
Lanham, MD 20706

Henry Greene
IIT Research Inst.
4600 Forbes Blvd.
Lanham, MD 20706

Mary Bender
U.S. Army CECOM
Advanced Softech Tech.
Ft. Monmouth, NJ

ABSTRACT

With 206 validated compilers, developers of real-time embedded systems need guidance in the selection of runtime environments (RTEs) to ensure that the RTEs used can meet strict timing and storage requirements. The purpose of this study was to assist these developers in the selection of RTEs by providing a means for prioritizing RTE elements and introducing the concept of a composite benchmark to evaluate candidate RTE performance. Specifically, this study prioritized RTE elements and developed preliminary requirements for a composite benchmark for one class of systems supported by the U.S. Army Communications-Electronics Command (CECOM), Ft. Monmouth, NJ. The class of systems studied was Communications and Electronic Intelligence (COMINT/ELINT).

This paper presents the several steps in the process developed to prioritize RTEs. These steps are system selection, prioritizing RTE elements and the use of the prioritized elements to prioritize groups of benchmarks. The purpose of a composite benchmark, the development process for such a benchmark, and a preliminary description of a composite benchmark for COMINT/ELINT systems are given. Recommendations for the use of benchmarks in RTE selection are made.

SYSTEM SELECTION OVERVIEW

The first step was to select a class of CECOM systems to analyze. At the time of this study CECOM supported 136 systems. It was impractical to study all of the systems because of the system diversity and the number of systems. Thus, the goal of the selection process was to identify a class of systems to study that would be most beneficial to CECOM, i.e., the class that contained the most real-time systems supported by CECOM. The specific systems selected were representative of the particular class.

The class of systems chosen were from the Battlefield Functional Area (BFA) of Intelligence Electronic Warfare (IEW); specifically, the class of COMINT/ELINT systems. The systems that represented the COMINT/ELINT class were Improved Guardrail V (IGRV), Communication High Accuracy Airborne Location System (CHALS), Advanced Quick Look (AQL), and Trailblazer B. These systems were chosen because they were large and complex real-time embedded systems. An assumption was made that if a method could be developed that prioritized RTE elements for large and complex real-time embedded systems the method could be applied to other real-time embedded systems.

PRIORITIZATION OF THE RTE ELEMENTS

The next step was to prioritize the RTE elements. A prioritization matrix was used to identify the critical RTE elements for COMINT/ELINT systems. The prioritized list of RTE elements was to be used to prioritize benchmarks. The axes of the prioritization matrix are embedded computer system features (the columns), and RTE elements (the rows).

THE SIX EMBEDDED COMPUTER SYSTEM (ECS) FEATURES

The Software Engineering Institute (SEI) determined that there are six basic features of an embedded real-time system: time control, concurrent control, I/O control, error handling, numeric computations, and internal representation [Weideman 1987]. The six features were developed from a definition, a general requirement, and basic characteristics of embedded computer systems. The six ECS features make up the columns of the prioritization matrix.

Each of the features has a weight assigned to it. The weights represent the relative importance of an ECS feature with respect to the class of systems being studied. The sum of the weights must equal 100%. The 100% signifies an entire system, and the

Support for this research was provided by AJPO Contract Number MDA903-87-D-0056 with funds provided by CECOM Center for Software Engineering.

separate weights indicate the importance of each feature to that system. The weights should not change as one moves from one system to another, provided one looks at systems in only one class. If the class of systems is changed, the weights will change.

Determining the weights for the COMINT/ELINT class involved two steps. The first was to understand which features were important and to begin to quantify their importance by studying the system requirements. For this study each requirement was mapped to the particular ECS feature to which it pertained. This step resulted in the majority of the requirements being mapped to I/O control.

This first step gave an indication of which features were important and what number to assign as its weight. It did not take into account issues that affect the performance of a system. The primary concern was with concurrent control and time control. The requirements may define the need for concurrency, but they do not represent the solution, which is the algorithm that is used to meet concurrency needs. Also, the requirements may define the time limits imposed on the system, but they do not reflect the stringency of those limits.

Step two was to adjust the weights by studying the requirements and determining their effect on the performance of the systems. Then, taking into account the results of step 1 and step 2, weights were subjectively assigned to each ECS feature (see Table 1).

Table 1
The Final Weights Assigned to the Features

Features	Weights
Concurrent Control	20%
Time Control	20%
I/O Control	25%
Error Handling	10%
Numeric Computation	10%
Internal Reproduction	15%

THE RTE ELEMENTS

The rows of the prioritization matrix are the eleven RTE elements. They were obtained from the document "A Framework For Describing The Ada Runtime Environment" [ARTEWG 1988]. These RTE elements are the following:

- Memory Management
- Processor Management
- Interrupt Management
- Time Management
- Exception Management
- Rendezvous Management

- Task Activation
- Task Termination
- I/O Management
- Commonly Called Code Sequences
- Target Housekeeping.

The RTE elements make up the rows for the prioritization matrix. These elements are assigned rates. The rating is for quantifying the effect that an RTE element has on the performance of an ECS feature. Rating an element against an ECS feature is independent of the class of systems of interest.

A rating scale is used to rate an element. The following scale was used in this prioritization matrix. Following the scale, each classification is defined.

Intrinsic = 9
Supportive = 5
Extrinsic = 1

Intrinsic (9): An RTE element that is foundational to the performance of a particular feature.

Supportive (5): An RTE element that, although not intrinsic, has a role in the performance of a particular feature.

Extrinsic (1): An RTE element that at most has a minor role in the performance of the particular feature.

The two documents previously cited were influential in the rating process: the ARTEWG document, "A Framework for Describing Ada Runtime Environments" [ARTEWG 1988] and the SEI document, "Ada for Embedded Systems: Issues and Questions" [Weideman 1987]. The rating process involved concentrating on one ECS feature to determine whether the element was intrinsic to the performance of the feature. If it was, a "9" was entered into the square. If it was not, the RTE element was determined to be either supportive or extrinsic.

APPLICATION OF THE RTE PRIORITIZATION MATRIX

After all the weights and rates had been determined the next step was to multiply the weights by the rates. This step integrated all of the components of the prioritization matrix: the ECS features, their relative importance to the class of systems (the weight), and the RTE elements' ratings.

The last step was to sum all the products in a given row. The result was a prioritized list of RTE elements. The element with the highest total for a row was the most critical element, and the element with the next highest was the next most critical, and so on.

THE MATRIX APPLIED TO THE COMINT/ELINT CLASS

Figure 1 presents the prioritization matrix for COMINT/ELINT systems.

The following is the prioritized list of RTE elements.

1. Memory management 700
2. Time management 660
3. I/O management 640
4. Processor management 560
5. Rendezvous management 560
6. Exception management 540
7. Interrupt management 500
8. Task Activation 380
9. Task Termination 380
10. Target Housekeeping 380
11. Commonly Called Code Sequences .280

This list is the driver for prioritizing groups of benchmarks.

PRIORITIZATION OF GROUPS OF BENCHMARKS

The last step was to map benchmarks to the RTE elements they measure. The combination of a prioritized list of RTE elements and benchmarks mapped to each element produced a prioritized list of groups of benchmarks. This is because for each element there was a group of benchmarks that mapped to it. Thus, it was the groups that were prioritized and not the individual benchmarks.

Some sources of benchmarks that can be mapped to RTE elements and thus prioritized are the Ada Compiler Evaluation Capability (ACEC) [Loa 1988] benchmarks, the Real-Time Performance Benchmarks for Ada [Goel 1988] and the Performance Issues Working Group (PIWG) benchmarks.

ECS FEATURES

	CONCURRENT CONTROL	TIME CONTROL	I/O CONTROL	ERROR HANDLING	NUMERIC COMPUTATIONS	INTERNAL REPRESENTATION	TOTAL
WEIGHTS	20	20	25	10	10	15	100
MEMORY MANAGEMENT	9 180	5 100	9 225	5 50	1 10	9 135	700
PROCESSOR MANAGEMENT	9 180	9 180	5 125	5 50	1 10	1 15	560
INTERRUPT MANAGEMENT	5 100	5 100	9 225	5 50	1 10	1 15	500
TIME MANAGEMENT	9 180	9 180	9 225	5 50	1 10	1 15	660
EXCEPTION MANAGEMENT	5 100	5 100	5 125	9 90	5 50	5 75	540
RENDEZVOUS MANAGEMENT	9 180	9 180	5 125	5 50	1 10	1 15	560
TASK ACTIVATION	9 180	5 100	1 25	5 50	1 10	1 15	380
TASK TERMINATION	9 180	5 100	1 25	5 50	1 10	1 15	380
I/O MANAGEMENT	5 100	9 180	9 225	5 50	1 10	5 75	640
CODE SEQUENCES	1 20	1 20	1 25	5 50	9 90	5 75	280
TARGET HOUSEKEEPING	1 20	5 100	1 25	5 50	5 50	9 135	380

RATING SCALE

INTRINSIC	9
SUPPORTIVE	5
EXTRINSIC	1

Figure 1. Prioritization Matrix for COMINT/ELINT Systems.

COMPOSITE BENCHMARK

The majority of benchmarks available either test a specific element of the RTE in isolation, or exercise several elements in some unspecified combination. What is needed is a single benchmark that tests elements of an RTE while interacting in a manner that is consistent with their interaction during actual system operation. Such a benchmark would evaluate an Ada RTE by forcing the RTE to perform operations that would mirror the operations performed by the system to be developed. A composite benchmark is such a model of the capabilities of a particular class of systems.

PURPOSE OF A COMPOSITE BENCHMARK

The purpose of a composite benchmark is to stress a computer and its RTE to check their ability to perform the capabilities of a particular class of systems. A composite benchmark allows a software developer to run one benchmark, which will give him a general idea of whether a particular RTE can perform the capabilities of a given class of systems. A composite benchmark tests each capability individually and, importantly, the interaction among the capabilities.

THE DEVELOPMENT OF A COMPOSITE BENCHMARK DESCRIPTION

Developing a composite benchmark description for a particular class of systems is not a simple task, but once developed it can be used to aid in the selection of an RTE for any system in the given class. The following three steps should be followed when developing composite benchmarks:

1. Identify the common capabilities of the particular class of systems by studying the requirements and functions of the systems within the class.
2. Define and analyze each capability. The description should include all functions common to the systems in the class. If a particular function is common to several of the systems within the class, it should be included in the description because the function may be performed in a new system being developed.
3. Document the interactions and interfaces among the capabilities in a format that facilitates computer program code development. When writing the description, there needs to be continuous interaction between the description writer and programmer to ensure that the composite benchmark will be accurate and understandable. The description writer must have an in-depth technical knowledge of the class of systems being studied.

COMPOSITE BENCHMARK FOR COMINT/ELINT SYSTEMS

A preliminary description for a composite benchmark was developed in this study for COMINT/ELINT systems. The goal was to develop the idea and an approach for developing a composite benchmark. The preliminary composite benchmark models the five capabilities of COMINT/ELINT systems: intercept, direction finding, emitter location, analysis, and reporting. This description was given to another company, TAMSCO, for code development.

RTE SELECTION PROCESS

The final phase of this study was to determine how the prioritized benchmarks and the composite benchmark should be used when selecting an RTE. It was determined that choosing an RTE is a three-step process. The first step is to eliminate all RTEs that cannot perform beyond a minimum required threshold in each area critical to system performance. The second step is to begin with the set of RTEs that satisfy the minimum threshold requirements and select the small set of RTEs that performs best in the areas critical to system performance. The final step is to compare the costs, the vendor support provided, and any other mitigating circumstances for the final selection of an RTE or compiler. The first two steps involve the use of benchmarks.

The composite benchmark is to be used to test the minimum threshold of RTEs. This means the developer would only have to run one benchmark to eliminate RTEs not suitable for his particular class of system. Then the other benchmarks would be used to test the remaining RTEs to see which RTEs perform the best in the areas critical to system performance. Because the RTE elements are prioritized, the critical areas and the benchmarks that measure those areas are known.

At this time the development of the composite benchmark is still in the preliminary phase. Until the composite benchmark matures, only the prioritized list of groups of benchmarks can be used to test RTEs.

CONCLUSION

The objective of this study was to provide software developers with guidance in the selection of a compiler and its RTE to ensure that all the timing and storage requirements of their real-time embedded systems can be met. Because there is no "universal best" RTE, the selection of an RTE is based on the needs of a specific class of system. This means the selection of an RTE is

domain specific, and the use of a prioritization matrix as described in this paper will allow for the prioritization of RTE elements for a particular domain. The prioritized RTE elements are then used to prioritize groups of benchmarks.

Another domain specific way to test RTEs is through the use of a composite benchmark. Composite benchmarks would test each candidate RTE and eliminate those that don't meet a set of minimum requirements. Unlike existing benchmarks, a composite benchmark will take into account the interactions and interfaces that go on within a system.

Thus, when selecting an RTE, the composite benchmark would be used to test the minimum threshold of RTEs. Then the prioritized groups of benchmarks would be used to test the critical RTE elements to determine which RTEs perform best in those critical areas.

When developing this method to prioritize RTE elements, two preliminary steps, mapping system capabilities to Ada constructs and mapping Ada constructs to Ada RTE elements were done. These steps are not discussed in this paper so the more significant results of the later steps can be highlighted. The objective of mapping the system capabilities of COMINT/ELINT systems to Ada constructs was to determine what Ada constructs would be used in real-time embedded systems. The results were that all Ada constructs would be used. The objective of mapping Ada constructs to Ada RTE elements was to determine which of the eleven RTE elements were not important in real-time embedded systems. During this step, one RTE element, target housekeeping, was determined not to be important. This finding however was contradicted after the implementation of the prioritization matrix. As a result the original assumption that target housekeeping was not important was reversed.

For the process of prioritizing RTE elements, it is recommended that after the particular domain (class of system) has been selected, the first step is the implementation of a prioritization matrix. If the matrix is applied to an ECS the rows and columns of the matrix are already in place. If the matrix is to be applied to any other type of system, a new set of column titles would have to be developed. This is because the features common for all ECSs might not be common to other systems, but there may be some overlap.

There are two future directions for this research. One is to mature the preliminary description of the composite benchmark into an in-depth specification and to develop the actual benchmark. Second is to apply this approach of prioritizing RTE elements and groups of benchmarks to another class of systems.

REFERENCES

ARTDAG. A Framework for Describing Ada Runtime Environment. SIGAda. 1988

Coel, Arvind Kumar. Real-Time Performance Benchmarks for Ada. Final Technical Report, Center for Software Engineering, CECOM, 1988

Leavitt, T., Terrell, K. Ada Compiler Evaluation Capability (ACEC) Version Description Document. AFWAL-TR-88-1093. Boeing Military Airplane for Air Force Wright Aeronautical Laboratories. 1988.

Weideman, Nelson, et al. Ada for Embedded Systems: Issues and Questions. Software Engineering Institute (SEI). Carnegie Mellon University. Pittsburgh, PA. 1987.



Ms. Sharon R. Iafkowitz is a Software Engineer with the IIT Research Institute (IITRI). Ms. Iafkowitz was the technical lead in research on establishing and evaluating runtime features of interest for real-time systems. Her research interests include real-time embedded systems, software application development, and database design and analysis. She received her B.S. degree from the University of Maryland in Information Systems Management.



Mr. Henry Greene is a Software Engineer with the IIT Research Institute (IITRI). His research interest are in Ada reusability and machine-independence characteristics. He is also interested in object oriented requirements and design with respect to real-time systems using Ada. Mr. Greene received his B.S. degree from Dordt College in Mechanical Engineering. He is currently working toward an M.S. degree in Computer Science at John Hopkins University.



Mrs. Mary E. Bender is a Computer Scientist with the Center for Software Engineering, U.S. Army CSECOM, Ft. Monmouth, N.J. She is the project leader for their technology program in Ada real-time applications and runtime environments. She received her B.A. degree in Computer Science from Rutgers University in Brunswick, N.J.

Real-time Performance Benchmarks For Ada

Arvind Goel

TAMSCO

Abstract: This paper describes the Ada benchmarking effort undertaken by the author under contract to Advanced Software Technology Directorate, Center for Software Engineering, US Army, Ft. Monmouth, NJ. Ada benchmarks have been developed to measure the performance of Ada compilers meant for real-time embedded systems. Three kinds of benchmarks have been developed: first kind measures the performance of individual features of Ada language important for real-time systems, second kind of benchmarks deal with determining the runtime system implementation in the areas of tasking, scheduling, memory management, exceptions, interrupt handling etc., and the third kind involves programming algorithms found in real-time embedded systems.

1. Introduction

The principal goal of Ada is to provide a language supporting modern software engineering principles to design and develop real-time embedded systems software. A motivating factor in the development of Ada as the Department of Defense standard language was the high cost of embedded system software development. Current Ada compiler implementations are unable to support these demands due to several reasons:

- they are written by software engineers with experience in large-system design,
- lack of operating system knowledge and real-time issues,
- more concern with passing the Ada Compiler Validation Capability Test suite,
- implementation and size of the Ada runtime system which differs widely from one compiler to another.

The performance and implementation approach of various Ada language features and the runtime system has to be benchmarked to assess an Ada compiler's suitability for a real-time embedded application. Ada benchmarking is much more complex from other languages because of the powerful and sophisticated runtime system that supports Ada features such as memory management, process scheduling and control, tasking, etc. and whose implementation varies from one compiler system to another. A benchmarking effort has been undertaken by TAMSCO to determine the

suitability of Ada compiler systems for embedded applications. Existing benchmarks have been researched and have been modified as necessary. New benchmarks were added as well as existing benchmarks were modified. The scope of this benchmarking effort is to determine

- the runtime performance of Ada features on a bare target system
- the runtime system implementations of various features of a particular Ada compiler system
- the performance of commonly used Ada real-time paradigms (also referred to as macro constructs)

The benchmarks developed have been run on a Verdix compiler targeted to a Motorola 68020 bare target. The results of running the benchmarks is stated in another report published by the author [1].

2. Ada Benchmarking

Ada benchmarking can be approached in 3 ways:

- design benchmarks to measure execution speed of individual features of the language,
- design benchmarks that determine among various other things implementation dependent attributes like the scheduling and storage management algorithms,
- design benchmarks that measure the performance of commonly used real-time Ada paradigms.

2.1 Measure Performance Of Individual Features

This approach measures the execution speed of individual features of the language and runtime system by isolating the feature to be measured to the finest extent possible. Such benchmarks are useful in understanding the efficiency of a specific feature of an Ada implementation. For example, a benchmark that measures the time for a simple rendezvous can be run on two Ada compiler systems. Based on the results, an application can choose one compiler system over the other. The problems with such an approach is determining and isolating the features of the language and runtime system that are important for real-time embedded system applications. Also, this approach requires a significant number of tests and the numbers produced have to be statistically evaluated to determine general performance.

2.2 Determining Runtime System Implementation

These benchmarks are concerned primarily with determining the implementation characteristics of an Ada Runtime System. The scheduling algorithm, storage allocation/deallocation algorithm, priority of rendezvous between two tasks without explicit priorities are some of the many implementation dependent characteristics that need to be known to determine if a compiler system is suitable for a particular real-time embedded application. Some implementation dependencies cannot be benchmarked and that information has to be obtained from the compiler vendor as well as the documentation supplied by the vendor. The ARTEWG document "Catalog of Ada Runtime Implementation Dependencies" [2] is a compiled list of features that are implementation dependent. This document has been consulted extensively in determining which implementation dependencies need to be benchmarked for real-time embedded systems.

2.3 Real-time Paradigms

There are a number of characteristics of real-time embedded systems that do not correspond to a specific Ada feature, but those characteristics can be constructed using a combination of Ada features. This approach also involves programming algorithms found in embedded systems. For example, a situation in real-time systems may be a producer that monitors a sensor and produces output asynchronously and sends it to a consumer. The producer task cannot wait for a rendezvous with the consumer (who might be doing something else) as the producer task might miss a sensor reading. To program this paradigm in Ada requires three tasks: a producer task, a buffer task that receives input from the producer task and sends the input to the consumer task. For real-time embedded systems, such paradigms can be identified and programmed in Ada. These benchmarks can be run on Ada compiler implementations and statistics gathered on their performance.

3. Microscopic Benchmarks

Microscopic benchmarks are designed to measure the performance of individual features of the Ada programming language. Benchmarks have been designed for all the major Ada language features that are important for real-time embedded systems. Since optimizing compilers generate different code for the same feature depending on the context in which the feature occurs, it has been attempted to benchmark a particular feature under different scenarios. The results will demonstrate the range of performance associated with a language feature.

After a detailed analysis of existing benchmark suites, it was determined that the methodology developed by the University of Michigan [1] is best suited for benchmarking specific Ada language and runtime features that are important for real-time embedded systems. This suite addresses the issues that are of concern when designing benchmarks some of which are: isolation of features, accuracy, and thwarting compiler optimizations[2].

3.1 Benchmark Timing

For benchmarks that measure time values using the system function `CLOCK`, the ideal design would be to determine the specific feature that needs to be measured and perform that feature sandwiched between calls to the system `CLOCK`. The difference in time is the execution time for that feature. For this measurement to be accurate, the resolution of the `CLOCK` should be considerably less than the time required by the operation to be measured. Generally, the system clock that are available to a benchmark designer may be accurate to a tenth of a second and that is inadequate to measure events in the millisecond and microsecond ranges. Some of the problems that have to be overcome when accessing the internal `CLOCK` function include:

1. **Clock Precision:** In designing portable benchmarks, one can only assume the presence of the function `CLOCK` in the package `CALENDAR`. If the precision of the `CLOCK` function (`SYSTEM.TICK`) is not very high it can cause errors in the timing measurements. Generally the execution time of a Ada feature is much smaller than `SYSTEM.TICK`.
2. **Clock Overhead:** Another problem is the inconsistent time required for the `CLOCK` function. Some compiler implementation return an aggregate data structure and this may require the calling of storage management functions resulting in inconsistent timing for the `CLOCK` function.
3. **Clock Jitter:** Clock readings are subject to the usual statistical variations associated with physical measurements and can be expected to show random variations known as jitter.

To overcome these problems, a technique known as the dual loop technique [2] is used to measure the execution time for a specific feature. In this technique an operation is performed repetitively, and the aggregate of multiple executions is timed. By performing the operation repetitively, the time duration of a test is increased and the system clock can measure this time precisely. In fact, this is done twice, once in a control loop without the feature being measured, once in a test loop with the feature. Subtracting the execution time of these two loops, and dividing by the number of executions yields a calculated time for one execution of the feature. The dual loop technique solves a number of problems that have been mentioned above.

3.2 Tasking Benchmarks

For Ada to fulfill its potential for embedded systems, its model of concurrency - the tasking model - must be sufficiently fast to meet the timing needs of such systems. Concern over efficiency and semantics of Ada tasking could force many organizations using Ada to avoid the tasking facilities entirely, relying instead on a separately written executive. There are specific concerns in the real-time application community regarding the semantics of the Ada tasking model and its potential implementation overhead. Ada tasking model is significantly different from current real-time paradigms such as cyclic executives. In fact translation of concurrency paradigms may force creation of intermediary tasks with the risk of compromising real-

time performance.

3.2.1 Task Activation/Termination Some points to note about task activation/termination benchmarks are:

1. The time to elaborate, activate and terminate a task is measured as one value. The individual components of the measurements are too quick to measure with the available CLOCK resolution.
2. Some implementations may implicitly deallocate the task storage space on return from a procedure or on exit from a block statement (when the task object is declared in a procedure or block statement). If task space is implicitly deallocated, the number of iterations can be increased to get greater accuracy for task activation/termination measurement. If task space is not deallocated on return from a procedure or block statement, then the attribute STORAGE_SIZE can be changed such that the number of iterations can be increased.

The first set of benchmarks measure task activation and termination time under various scenarios:

- These benchmarks measure task activation/termination timings for task objects declared in block statements, procedures, packages, other tasks, arrays of tasks, and as part of a record. The effect of existing active tasks on task activation/termination timings is also determined.
- These benchmarks measure task activation/termination timings for task objects created via the new allocator. Timings are measured for tasks created in a block statement, procedure, and array of tasks. The effect of existing active tasks on task activation/termination timings of tasks created via the new allocator is also determined. Since access object does not exist on exit from the block statement, the timing measured includes both allocation and deallocation timings for the task as well as task activation and termination times.

Some conclusions that can be drawn are:

1. The activation and termination time of tasks for the various scenarios that are described above determines if a real-time programmer should declare tasks for time-critical modules in packages or in the main procedure, in procedures that are repeatedly called by other procedures, or within other tasks in the system.
2. If an implementation does not deallocate the storage space occupied by a task on exit from a procedure then the timings for task activation/termination using arrays and without arrays should be compatible otherwise the timings for task activation/termination using arrays can be significantly higher.

3.3 Task Synchronization

In Ada, tasks communicate with each other via the rendezvous mechanism. Rendezvous are effectively similar to procedure calls, yet they are much more complex to implement, and therefore create a tremendous amount of overhead for the run-time system. This

overhead affects the efficiency of the system in both sizing and timing. Because of the timing constraints in a real-time embedded system, it is essential that the rendezvous mechanism be as efficient as possible.

3.3.1 Simple Rendezvous The simple rendezvous time gives a lower bound on the rendezvous time because no extraneous units of execution are competing for the CPU. This overhead is expected to occur each time two tasks are in a rendezvous and does not include any execution time for the statements within the accept body. Simple rendezvous benchmarks:

- Measure time for simple rendezvous where entry calls with no parameters are made to tasks declared in the main program, block statements, packages, procedures. Benchmarks are also designed to determine if it is advantageous for an application to have more tasks with less entries or less tasks with more entries.
- Benchmarks have also been designed to determine the affect on entry call time as the number of accept alternatives in a select statement increases. For some implementations, time for a rendezvous may also be affected by the position of the accept alternative in the select statement. Based on these tests, application designers can choose to place the most time-critical accept statements in a certain manner.
- Measure the affect of guards (on accept statements) on rendezvous time, where the main program calls an entry in another task (with no parameters) as the number of accept alternatives in the select statement increases.

3.3.2 Complex Rendezvous Complex rendezvous benchmarks:

- Measure the time required for a complex rendezvous, where a procedure in the main program calls an entry in another task with different type, number and mode of the parameters. Rendezvous time may depend on the size and type of the passed parameters which may involve both the task stacks or the allocation of a separate area for passing large structures. Increasing rendezvous times for array parameters as the size of the array increases implies that the implementation uses pass by copy instead of pass by reference.
- Determine the affect on time required for a complex rendezvous, where the main program calls an entry in another task with different type, number and mode of the parameters as the number of accept alternatives in the select statement increase. For some implementations, time for a rendezvous may also be affected by the position of the accept alternative in the select statement. Also, the time for rendezvous may increase as the number of integer parameters passed during the rendezvous increases.
- Determine the cost of using the terminate option in a select statement. If the overhead due to the terminate option is high, then this option should not be used especially if the selective wait is inside a loop.
- Determine the overhead due to conditional and timed

entry calls when a) the rendezvous is completed b) the rendezvous is not completed. This benchmark measures the execution time overhead of the conditional and timed entry calls when the rendezvous does and does not take place. This overhead has to be considered whenever polling is used to establish synchronization between tasks.

- Measure the affect on time required for a complex rendezvous, where a procedure in the main program calls an entry as the number of activated tasks in the system increases. Time for rendezvous can degrade with the number of eligible tasks due to the search and sorting involved with prioritized dispatching.

3.4 Scheduling and Delay Statement

Task scheduling is an important consideration for a multitasking application. Real-time embedded systems contain jobs with hard deadlines for their execution. Failure to meet a deadline reduces the value of the job's execution possibly to the extent of jeopardizing the system's mission. It is the responsibility of the runtime system's scheduling mechanism to guarantee that the most important deadlines are met while also meeting as many of the less important deadlines as possible. For scheduling tasks at a particular time, the delay statement can be used in conjunction with the CALENDAR package. The precision of the timing depends on the implementation of the package CALENDAR and on the granularity of the underlying scheduler. The semantics for the delay statement, however, provides only that the delay specified is a minimum amount of the delay time. For real-time embedded systems, it is the maximum delay not the minimum delay which is of interest. Another reason why Ada implementation of periodic tasks is not reliable is the possibility of an interrupt between the time the delay is computed and the time the delay is requested. Hence the time at which delay expires cannot, in general, be predicted in advance.

- Determine the minimum delay time. This benchmark determines the actual delay time for a desired delay time specified in the delay statement. This benchmark starts by calculating the actual delay time for a minimum delay of DURATION'SMALL. The desired delay time is increased in steps and the actual delay time calculated.
- Determine if user tasks are pre-emptive. Does a completed delay interrupt the currently executing task to allow the scheduler to select the highest priority tasks.

3.5 Memory Management

Ada is the first high order language intended for mission critical, real-time applications that requires dynamic memory allocation and deallocation. The amount of storage required in these circumstances cannot be determined by static examination of a program and benchmarks must be executed to determine the efficiency of an implementation's storage utilization.

- These benchmarks determine the time for allocating storage known at compile time. Time is measured to allocate and deallocate a fixed amount of storage upon entering a subprogram or a declare block. The

size of the objects is known at compilation time, but space for the objects is allocated on the stack at runtime.

- Measure time for allocating variable amount of storage. Variable storage allocation involves allocation of a variable amount of storage when entering a subprogram or declare block. In this test case, arrays of different dimensions bounded by variables are allocated and the size of the objects is not known at compilation time. Tests are also designed to determine the threshold when objects are allocated from the heap rather than on the stack.
- Memory Allocation via the New Allocator. Based on these timing measurements, real-time programmers can decide whether to use the new allocator for object elaboration or to declare the object as in fixed length case.
- Determine the effect on time required for dynamic memory allocation when memory is continuously allocated without being freed. Time for dynamic allocation can depend on the state of storage management following previous allocations due to the need to recover storage and efficiently manage the available space. If memory is allocated in a loop via the new allocator, and the memory that is allocated is not freed, then the time required for dynamic memory allocation can be affected as more space is allocated.
- Determine the effect on time required for dynamic memory allocation when memory is continuously allocated without being freed and also as the number of tasks in the system increases.

3.6 Exceptions

Real-time embedded systems should be able to handle unexpected errors at run-time. Unexpected errors could have disastrous consequences if not handled properly. Many real-time systems operate for long periods of time in stand alone mode and there is a need for efficient and extensive error-handling for such systems. The Ada exception handling mechanism provides a means by which errors can be detected and reported without catastrophic results. Two kinds of exceptions can be raised when a real-time embedded system is running: a) user-defined Ada exceptions and b) predefined Ada exceptions (like NUMERIC_ERROR, CONSTRAINT_ERROR, TASKING_ERROR, etc.). Predefined exceptions may also be raised because particular conditions are detected in the underlying computing resource. Exception Benchmarks:

- Measure the overhead associated with a code sequence that has an exception handler associated with it, yet no exception is raised during the execution of that code. Since exceptions are used to indicate "exceptional situations", exception handlers should not be executed during normal program execution.
- Measure exception response time for a) user-defined exception and b) pre-defined exceptions NUMERIC_ERROR, CONSTRAINT_ERROR, and TASKING_ERROR raised both by the raise

statement as well as due to abnormal situations in the application code. If the exception handling times are significant, real-time embedded system programmers can check and react to error situations in their program body rather than using using Ada exceptions.

- Measure a) timing overhead due to exceptions and b) exception response time when exception handled in the block statement when additional tasks are present in the system. Real-time embedded systems have multiple tasks existing in the system. The times measured by these benchmarks determine the affect of multiple existing tasks on exception response times and timing overhead due to exceptions.
- Measure Exception handling time when exception is raised and propagated one (two, three) level(s) below where it is handled. User-defined, and pre-defined (CONSTRAINT_ERROR, NUMERIC_ERROR, and TASKING_ERROR) exceptions are raised via the raise statement as well as abnormal situations in code. There are three scenarios: a) no idle tasks exist in the system when the exceptions are raised, b) 5 idle tasks exist in the system when the exceptions are raised and c) 10 idle tasks exist in the system when the exceptions are raised.
- Measure time to handle TASKING_ERROR exception in the calling task. This benchmark is executed with 3 scenarios: none (5, 10) idle tasks existing in the system when the exception is raised. If task exception handling time within a rendezvous is costly when compared to exception handling time in a procedure or block, then serious consideration must be given to providing an exception handler within the accept body of the time-critical tasks.
- Measure time to propagate and handle exception when a child task has an error during its elaboration. This benchmark is executed with 3 scenarios: none (5, 10) idle tasks existing in the system when the exception is raised.

3.7 Chapter 13 Benchmarks

Ada defines some features which allow a programmer to specify the physical representation of an entity, i.e., map the abstract program entity to physical hardware. These features are implementation-dependent: an implementation is not required to support these features. For real-time embedded systems, it is necessary to that the Ada LRM Chapter 13 features be implemented and made mandatory as the application designers have to deal with two levels, the abstract and representation level. Some Chapter 13 benchmarks include:

- These benchmarks measure time to perform standard boolean operations (XOR, NOT, OR, AND) and assignment and comparison operations on records and arrays of booleans. The tests are performed on entire arrays as well as components of arrays. The arrays are PACKED with the pragma 'PACK', representation clause is used to specify array size and the arrays are NOT PACKED with the pragma 'PACK'.

- Measure the time to do an unchecked conversion of different types of objects to other types.

- Measure the time to store and extract bit fields using Boolean and Integer record components. There are 3 scenarios:

- The time to store and extract bit fields that are NOT defined by representation clauses.
- The time to store and extract bit fields that are defined by representation clauses.
- The time to store and extract bit fields that are packed by PRAGMA PACK.

- Measure the time to perform a change of representation from one record representation to another. Measure the time to perform a change of representation from a packed array to an unpacked array.

- Measure the time to perform POS, SUCC, and PRED operations on enumeration type with representation clause specification.

3.8 Interrupt Handling

In real-time embedded systems, efficient handling of interrupts is very important. Interrupts are asynchronous events. In a real-time embedded system, interrupts are critical to the ability of the system to respond to real-time events and perform its required functions and it is essential that the system responds to the interrupt in some fixed amount of time. Benchmarks for interrupt handling include:

- Measure Interrupt Response Time. Techniques for measuring interrupt response time are very difficult as hardware external to the CPU must be involved in order to generate interrupts. This measure is totally dependent on the hardware involved, although some general criteria for measuring the interrupt response time is discussed in this report. External instrumentation (e.g., electronic equipments, real-time timers etc.) is required to accurately capture the time of interrupt occurrence.

3.9 Clock Function and TYPE Duration

For real-time embedded systems, the CLOCK function in the package CALENDAR is going to be used extensively. The implementation of the system clock is an important factor in the overall capabilities of the system. The CLOCK function reads the underlying timer provided by the system and returns the value associated with the timer. If the time taken to execute the CLOCK function is less than the time resolution, successive evaluations of CLOCK will return the same value. Most computer architectures have two kinds of hardware for timing. The counter timer chip used to drive the system clock defines the minimum granularity of time available to the system. The second level of granularity is the basic clock period which can be found in the Ada package SYSTEM (SYSTEM.TICK). Typically, some reasonable value is chosen for the size of the CLOCK period, and an interrupt is generated at this rate.

The Ada type DURATION is not required to have the same resolution as the clock period. It is required by the Ada LRM to be at most 20 milliseconds and that it be no more than 50 microseconds. A real-time embedded system has timing constraints that require response within a predetermined time interval. The clock period or resolution of type DURATION must support these requirements. Benchmarks include:

- Measure CLOCK function overhead. If the overhead associated with executing the CLOCK function is high, then real-time embedded systems will be hesitant to use the CLOCK function. Also, as discussed previously, the CLOCK overhead does add to the time required to make a benchmark measurement. But the dual loop benchmarking strategy can negate this effect by subtracting the control loop from the test loop.
- Measure CLOCK resolution. If the resolution time of the CLOCK function is not high, then for real-time applications a higher resolution clock is needed.

3.10 Numeric Computation

An embedded system must be able to represent real-world entities and quantities to perform related manipulations and computations. There should be support for numerical computation, units of measure (including time), and calculations and formulae from physics, chemistry etc. Benchmarks include:

- Measure the overhead associated with a call to and return from the "+" and "-" functions provided in the package CALENDAR. For real-time embedded systems, it is necessary to dynamically compute values of type TIME and DURATION. If the overhead involved in this computation is significant, the actual delay experienced will be longer than anticipated which could be critical for real-time systems.
- Determine time required for float matrix multiplication and addition, factorial and square root calculations.

3.11 Subprogram Overhead

In Ada, subprograms rank high among program units from a system structure point of view. Systems designed and implemented in Ada appear as a collection of packages and subprogram units, each of which may have multiple procedures. For real-time programmers to use good programming techniques and structured system design methodologies, it is important that subprogram call mechanism be as efficient as possible.

If the subprogram overhead is high, then the compiler can generate INLINE expansion at the cost of increasing the size of the object code. However, if calls to that subprogram are made from a lot of places, then the pragma INLINE defeats the purpose due to increase in size of object code. In embedded systems where memory is at a premium using pragma INLINE may not be a practical solution. Also, a compiler implementation may not support pragma INLINE. If subprogram overhead is high, programmers may be forced to use assembly language for time critical regions.

From our own experience as well as after analyzing existing benchmarks, subprogram overhead has to be measured for inter- and intra-packages as well as generic and non-generic instantiations of code. Procedure Call Latency is the elapsed time between the moment of the event to the start of the statement execution following the event. In all the benchmarks, simple and composite parameters are passed with modes in, out, and in out. The following cases are considered:

- Intra package reference: both caller and called subprogram are part of the same package. Procedure call overhead is also measured for intra package calls with pragma INLINE. If the timing for subprogram overhead for intra-package calls (without pragma INLINE) is nearly zero, then it is possible that the compiler is INLINING procedure calls.
- Inter Package Reference: The motivation for inter-package tests is to compare the subprogram call overhead and procedure call latency time between intra- and inter-package calls.
- Instantiations of Generic Code: In the tests for inter- and intra-package calls, the subprograms are part of generic packages that are instantiated.

3.12 Pragmas

The main purpose of pragmas is to select particular runtime features of the language or to override the compiler's default. There are certain predefined pragmas which are expected to have an impact on the execution time and space of a program. These include: CONTROLLED, INLINE, OPTIMIZE, PACK, PRIORITY, SHARED, and SUPPRESS.

- The benchmarks for pragma SUPPRESS determine the improvement in execution time when pragmas SUPPRESS is used. These are test problems which contain the same source text where the only difference between the problems is the presence (or absence) of pragmas. Pragma SUPPRESS causes the compiler to omit the corresponding exception checking (RANGE_CHECK, STORAGE_CHECK etc.) that occurs at runtime.
- Determine if pragma CONTROLLED has any affect for a access type object.
- Benchmarks for pragma INLINE and PACK are covered before.

3.13 Input/Output

Embedded systems depend heavily on real-time input and output. An Ada embedded system must have potential access to I/O ports, to control, status and data registers (for a memory mapped scheme), to direct memory access controllers, and to a mechanism for enabling and disabling interrupts. An excellent discussion of I/O is provided in the paper by Weideman [9]. Real-time I/O is subject to strict timing requirements and can be either synchronous or asynchronous. To handle I/O for a specialized device, a special interface is needed. This interface provides the attributes found in device drivers and interrupt handlers. I/O benchmarks:

- Determine if true asynchronous I/O is implemented.
- These benchmarks deal with TEXT IO. The tests are designed to open data file for reading and copying the data to another file. Time is measured to achieve the above for each type of IO mentioned above. create an output file and then copy the fixed type values from the input file to the output file.

4. Runtime Implementation Benchmarks

The Ada Language Reference Manual (LRM) has a lot of implementation dependent features that are of concern to real-time programmers. A list of the implementation dependent features is compiled in a document published by the Ada Runtime Environment Working Group [5]. The large variance in implementation options for a feature affect application program behavior and efficiency. This is a clear signal that simply adopting the language as defined in the LRM is not enough for real-time embedded systems. The implementation approach of various Ada language features and the runtime system has to be benchmarked to assess an Ada compiler's suitability for a real-time embedded application.

4.1 Tasking

Tasking runtime implementation dependencies:

- Determine if task space is deallocated on return from a procedure (when a task that has been allocated via the new operator in that procedure terminates). In real-time embedded systems, where space is at a premium, it is necessary that task space be deallocated when that task terminates.
- Determine if tasks that are allocated dynamically by the execution of an allocator do not have their space reclaimed upon termination when access type is declared in a library unit or outermost scope. It might be impossible for the runtime system to deallocate the task storage space after termination. This is because the access value might have been copied and an object might still be referencing the terminated task's task control block.
- Determine the order of elaboration when several tasks are activated in parallel. When several tasks are activated in parallel, the order of their elaboration may affect program execution.
- Can a task, following its activation but prior to the completion of activation of tasks declared in the same declarative part, continue execution. The activation of tasks proceeds in parallel. Correct execution of a program may depend on a task continuing execution after its activation is completed but before all other tasks activated in parallel have completed their respective activations.
- If the allocation of a task object raises the exception STORAGE_ERROR, when is the exception raised? The LRM does not define when STORAGE_ERROR must be raised should a task object exceed the storage allocation of its creator or master. The exception must be no later than task activation; however an implementation may choose to raise it earlier.
- What happens to tasks declared in a library package when the main program terminates? For some real-time embedded applications, it is desirable that such tasks do not terminate. System designers need to know this information.
- Determine order of evaluation of tasks named in an abort statement. Abort statement provides a convenient way to terminate a task hierarchy. When a task T1 aborts a task T2, the result T2'COMPLETED is true when evaluated by T1. Other tasks may not immediately detect that T2'COMPLETED is true. In real-time embedded systems, tasks may have to be aborted in a certain sequence. The semantics of the abort statement do not guarantee immediate completion of the named task. Completion must happen no later than when the task reaches a synchronization point.
- Some other runtime implementation dependencies that concern the abort statement and cannot be benchmarked are:
 - When does a task that becomes aborted become completed?
 - What are the results if a task is aborted while updating a variable?
- Determine algorithm used when choosing among branches of a selective wait statement.
- Determine algorithm used when choosing among branches of a selective wait statement.
- Determine that on queued entry calls if a compiler uses the FIFO method of accepting the entry calls that arrived first, irrespective of the priorities of the entry calls queued up.
- Determine the order of evaluation for guard conditions in a selective wait.
- Determine method used to select from delay alternatives of the same delay in a selective wait.
- The following information needs to be supplied by the compiler vendors about task priority.
 1. Determine priority of tasks (and of the main program) that have no defined priority.
 2. Determine priority of a rendezvous between two tasks without explicit priorities.
 3. Determine if a low priority task activation could result in a very long suspension of a high priority task.
- Does delay 0.0 simply return control to the calling task or causes scheduling of another task.

4.2 Memory Management

- Determine STORAGE_ERROR threshold. This tests are basically concerned with determining at which point exception STORAGE_ERROR is raised. If memory is allocated in a loop via the new allocator, and the access variable that is pointing to the allocated memory remains throughout the run, then STORAGE_ERROR will be raised at some point. A

real-time embedded systems programmer needs to know the amount of memory that can be dynamically allocated without raising `STORAGE_ERROR`.

- Determine if Garbage collection is performed on the fly. Determine if Garbage collection is performed on scope exit.
- Determine if Unchecked Deallocation is implemented.

4.3 Interrupt Handling

The following information about interrupt handling is needed by the software designers. This information has to be obtained from the compiler vendor.

- Determine if an interrupt entry call is implemented as a normal Ada entry call, a timed entry call, or a conditional entry call. Implementation restrictions on these interrupt entries. Can they be called from the application code? Can they have parameters?
- Determine if an interrupt is lost when an interrupt is being handled and another interrupt is received from the same device.
- Determine the restrictions imposed by an implementation for selection of the terminate alternative that may appear in the same select statement with an accept alternate for an interrupt entry. Selecting the terminate alternative may complete the task which contains the only accept statements which can handle the interrupt entry calls, leaving the hardware unserved.
- Determine if an interrupt entry call invokes any scheduling decisions.

An interrupt need not invoke any scheduling actions.

- Determine if accept statement executes at the priority of the hardware interrupt, and if priority is reduced once a synchronization point is reached following the completion of accept statement.
- Determine if interrupt entries can be called from application code.

5. Real-Time Paradigms

The designers of real-time embedded systems have to live with the problems of Ada until a solution is found (maybe by revising the language). Users, system programmers, and academicians have found a number of useful paradigms for building concurrency. Real-time systems will be designed as a set of cooperating concurrent processes (Ada tasks) using the Ada tasking model. Translation of concurrency paradigms may force the creation of intermediary tasks with the risk of compromising real-time performance. This includes intermediary tasks, monitor/process structure, asynchronous message passing, interrupt procedures, and event signaling. These paradigms can be coded in Ada and benchmarked. Also, a compiler implementation may recognize these paradigms and perform optimizations to implement that paradigm much more efficiently. Some paradigms that have been benchmarked include:

- **Intermediary Tasks:** Many real-time implementations require buffered and unsynchronized communication between tasks. Rendezvous is the mechanism used in Ada for task communication. Due to the rendezvous being a synchronous and unbuffered message passing operation, intermediary tasks are needed to uncouple the task interaction to allow tasks more independence and increase the amount of concurrency. Various combinations of intermediary tasks are used in different task paradigms to create varying degrees of asynchronism between a producer and consumer. Intermediary tasks introduce a lot more rendezvous in a real-time system than if a producer and consumer were directly communicating with each other. The use of intermediaries also adds to the cost of executing a real-time design in Ada. The benchmarks in this section evaluate the cost of introducing intermediary tasks for various real-time tasking paradigms. The goal of these benchmarks is to give real-time programmers a feel for the cost of using such paradigms in a real-time embedded application and to avoid using such paradigms if the cost is unacceptable for a real-time system. Some of the scenarios that have been benchmarked include: Producer-Consumer, Buffer Task, Use of a Buffer and Transporter, use of a Buffer and Two Transporters, use of a Relay etc.

- **Asynchronous Exceptions:** Quick restarts of tasks are required in a number of real-time embedded systems. Ada model of concurrency does not provide an abstraction where a task may be asynchronously notified that it must change its current execution state. One way to implement asynchronous change in control is to abort the task and then replace it with a new one. Aborting a task may not be appropriate for an application because an abort can take a long time to complete or because the asynchronous change of control needed is something other than termination. Abort and task initialization are expensive operations and a abort could take a long elapsed time to complete.
- **Selection of Highest Priority Client during an Entry Call:** The LRM states that in a select statement if more than one accept is open and ready for a rendezvous, then any one accept can be chosen and the choice is left to the compiler implementor. In real-time embedded systems, it may be necessary to choose the highest priority waiting client.

This benchmark implements a generic package that orders client requests so that they are processed by the server in a priority order. This package logically exists as an intermediary between the clients and the server. The overhead to this solution is three additional rendezvous for each prioritized rendezvous.

- **Monitor/Process Structure:** A monitor is commonly used for controlling a systems resource. Such a task performs a watchdog function and would be classified as an actor task (Actor tasks are active in nature and make use of other tasks to complete their function). Semaphores are an effective low-level

synchronizing primitive. However, the use of semaphores in an complex application can result in disaster if an occurrence of a semaphore operation is omitted somewhere in the system or if the use of a semaphore is erroneous. A monitor replaces the need to perform operations on semaphores. Entry to a monitor by one process excludes entry by any other process. A monitor thereby ensures that if it has exclusive access to a resource, then a monitor's user has exclusive access to that resource. In this program, a monitor is developed in Ada. The problem is having a pool of data common to a group of processes. The data in the pool may be set by one or more processes or used by one or more processes. Any number of processes are allowed to read the pool simultaneously, but no reads are permitted during a write operation. The monitor developed is used to control the reading and writing of data to the pool.

- Mailbox: In message passing, a question that arises is where messages are to be deposited. A common paradigm involves "mailboxes" which are global variables updated by processes to provide asynchronous communication. These are specially suitable for such situations as the producer/consumer scenario in which a producer produces some output which is consumed by a consumer process. The mailbox implementation of this involves a global mailbox visible to both these processes, and a send operation by the producer into this mailbox. The consumer then performs a receive operation on the mailbox to retrieve the data.

6. Conclusions

Benchmarking Ada implementations to determine their suitability for real-time embedded systems is an extremely complex task. This job is made even more difficult due to differing requirements of various real-time applications. In the near future, the authors plan to develop composite benchmarks to model some real-time systems used in the US Army.

REFERENCES

- [1] A. Goel, "Real-time Performance Benchmarks For Ada", TAMSCO Technical Report, October, 1988.
- [2] R.M. Clapp et al., "Towards Real-time Performance Benchmarks for Ada", CACM, Vol. 29, No. 8, August 1986.
- [3] N. Altman, "Factors Causing Unexpected Variations in Ada Benchmarks", Technical Report, CMU/SEI-87-TR-22, October 1987.
- [4] N. Altman et al., "Timing Variation in Dual Loop Benchmarks", Technical Report, CMU/SEI-87-TR-21, October 1987.
- [5] "Catalogue of Ada Runtime Implementation Dependencies", ARTEWG Report, November, 1986.

- [6] "Catalogue of Interface Features and Options for the Ada Run Time Environment", ARTEWG Report, October, 1986.
- [7] "Technology Insertion For Real-time Embedded Systems", Labtek Inc., July, 1986.
- [8] "User's Manual For the Prototype Ada Compiler Evaluation Capability (ACEC)", Institute For Defense Analysis, October, 1985.

About the Author:

Arvind Goel received his B. Tech. degree in Electrical Engineering from IIT, Kanpur, India in 1980 and MS degree in Computer Sciences from the University of Delaware in 1982. He is the founder of Unixpros Inc. where he is working on developing composite benchmarks to model a class of real-time systems. He is also working on CASE tools and their application to real-time systems. His interests include programming languages, Ada compiler evaluation, APSE research and evaluation, and developing software for embedded applications and distributed targets.

Mailing Address:

Unixpros Inc.

16 Birch Lane
Colts Neck, NJ 07722

REAL-TIME Ada DEMONSTRATION PROJECT

Mary E. Bender
U.S.. Army CECOM
Center for Software Engineering
Ft. Monmouth, NJ

Thomas E. Griest
LabTek Corporation
8 Lunar Drive
Woodbridge, CT

ABSTRACT

The Ada programming language has been available to software developers for several years, yet its acceptance into the real-time embedded applications for which it was intended has been less than universal. This project was designed to study the capabilities of Ada in the most difficult real-time applications which have traditionally been done in low level languages. A compiler with essentially all of the optional features of Ada and very good tasking performance was selected to assess the state of the art in Ada compilation systems. The project is described as well as specific real-time requirements that were imposed on its implementation. Details of the problems encountered and the findings of the development team are provided to guide others who will be using Ada for similar applications in the near future.

BACKGROUND

There is an ongoing program at the Center for Software Engineering at CECOM to explore Ada real-time/runtime technology for the purpose of providing guidance to the developers of Army embedded real-time Ada systems. A large number of research tasks on various real-time topics have been completed and others are in progress. Work in this program area has been supported by CECOM, STARS, and AJPO.

The program is based on recognized problems and the consensus of experts in this area. It began with an extensive study to identify the root problems causing difficulties in the development of real-time systems written in Ada. Program managers and developers were interviewed and a list of problems was defined, analyzed, and compiled into a database. From this identified set of problems, studies were initiated on various topics. These

included tasks that investigated guidelines to select and use an Ada Runtime Environment (RTE), an approach to tailor and configure a RTE, benchmark evaluation and development for performance testing, reuse handbook extension for real-time, a real-time methodology framework, and transportability guidelines.

After establishing this set of problems, the next step was to pick a prominent difficulty and show how to solve it. A large percentage of those interviewed pointed to the lack of performance provided by an Ada RTE compared to what is needed for real-time embedded systems. In particular, Ada tasking facilities have performed poorly in comparison with alternative approaches. It should be noted that performance is always an issue in real-time systems, even when programmed in assembly language, but the problem is made more pronounced when a high order language such as Ada is used.

APPROACH

An approach to solving this performance problem was defined through the demonstration project which is the subject of this paper. It proposed to develop a real-time application with a single Ada program containing multiple tasks and measure its performance. Then the program would be distributed as appropriate onto multiple CPUs to obtain the desired performance that couldn't be obtained with a single CPU. This approach recognizes that to regain the performance lost through the use of a high order language, users must take full advantage of the language which allows complexity to be managed more easily. Additional complexity can take the form of a sophisticated algorithm which is more efficient or by adding additional processing elements to increase throughput. In order to improve the performance of systems developed with Ada, developers should

take advantage of parallel structures within the language which facilitate the use of additional processors to increase system performance. Having the ability to add processors to achieve system performance requirements allows for substantial risk reduction.

In addition to solving an identified problem, the overall goals of the project were many. The difficulties in real-time Ada programming were addressed from an Ada technology perspective. The demonstration wanted to show how to work through a problem and not work around it as may be necessary in the "heat of battle" associated with hardware/software integration. The project wanted to show there can be near-term solutions to critical problems while continuing research on long-term solutions. The results are intended to provide accurate details on some of the "perceived" problems with Ada for real-time to determine if they are "real" problems or rather problems created, for example, because of the preconceived mindset of the developer or other non-Ada causes. It was also designed to study and document difficulties in distributing tasks within Ada programs onto a multiple CPU RTE. It indicates what is achievable using Ada and how performance can be improved by judicious use of language features. Finally, the project attempts to show that it can be practical to use distributed systems effectively within the Ada model of concurrency and that the difficulty of adding additional processors can be minimized. The work done on this project is expected to be made available to other researchers, developers, compiler writers, and members of the Ada real-time community to aid in understanding and resolving the real-time Ada issues.

Two separate but related topics are covered by this project: a real-time application developed to be run on a uniprocessor, and the distribution of Ada programs for loosely coupled multiprocessors. The software was developed to be independent of the target architecture. Therefore it was developed with the intention of running on a single CPU essentially divorced from the underlying implementation architecture. The distribution aspects were intentionally deferred until after the detailed design to prevent the characteristics of the distribution mechanism from influencing the design. The intention was to make the distribution process as simple as possible. Current approaches require using non-Ada tasking primitives. Using non-Ada tasking primitives means modifying the source substantially when distribution occurs and understanding two distinct tasking models. This project utilized a

commercially available Ada compiler and supported the Ada semantics by extending the runtime system. The vendor supplied runtime code was not modified except to customize interface routines for a specific hardware timer and interrupt controller. The unit of distribution supported was the Ada task. The issue of shared memory has frequently been addressed by totally restricting its use. Although distributed shared variables weren't needed for this initial prototype, an analysis was done on what would be required to support such variables because they would give greater generality to what can reasonably be distributed. Full Ada semantics including the Ada rendezvous were preserved across the distributed system so that no source code changes were necessary to alter the allocation of tasks to processors. This allocation was done using a simple distribution table that specified the object names, the processor ID, and relevant characteristics.

PROJECT DESCRIPTION

The project involves the development of a typical weapon system application with severe performance requirements. The application is synthetic, but resembles many similar weapon systems in DoD applications in terms of scheduling and real-time requirements. It includes target tracking, weapon guidance, graphics and user interface functions all integrated into a complex application. In some cases, simplifications were adopted because they did not alter the nature of the application significantly and to reduce some of the detail that was felt to be redundant for demonstration of capability. The scenario chosen describes the problem and solution in terms recognizable to many users and developers of real-time applications without being tied to any one particular system.

The hypothetical weapons application is called the Border Defense System (BDS). It is designed to provide short to medium range protection against a massive armored attack. The BDS tracks ground targets and attempts to destroy these targets with guided rockets. In addition, a simulator was developed that provided target and rocket motion. The BDS receives target position information from a surveillance system (simulator), generates a real-time graphics display for an operator, launches rockets to intercept targets, provides real-time rocket guidance data, updates the color graphics display to indicate the rockets' flight progress in real-time, and provides post attack assessment information and the number of active targets and

rockets to the BDS operator.

The performance requirements of the BDS specify a one hundred percent hit rate while operating in the absence of effective countermeasure and with the conditions specified in the Parameter Data Base (PDB). The PDB enumerates target and rocket parameter ranges for factors such as velocity, turn-rate, thrust, and position. The BDS specification also states that the software shall be developed in the "full" Ada language. No assembly language is permitted. Ada "code" statements may be used, but are limited to a total of fifty. All application concurrency is expressed using the Ada tasking model (rendezvous) exclusively. The system includes heavy computational requirements such as square roots, tangent, and arc tangent, and rocket/target correlation. Also communication with the underlying network implementation is stressed.

The BDS is a hard deadline driven application - failure to meet timing requirements will result in mission failure. The total system demand is a combination of rocket guidance, graphics display update, and operator interface requirements.

PROJECT IMPLEMENTATION

The implementation of the demonstration project was accomplished by a team consisting of contractor and government personnel. The BDS and distribution effort was done by the contractor with the simulator being developed by the government. The development was at geographically separate locations. The contractor's facilities were approximately 150 miles from the government installation. Although frequent trips and phone conversations were used as well as electronic mail, the coherency of a complex design is difficult to maintain in this type of environment. High speed modems were installed to reduce computer-to-computer delays and shared access to a development system. Even with these services, it was generally felt that the project suffered from communication errors. Due caution should be exercised whenever multiple design sites are planned for software development.

The design approach is time/risk driven to address the areas first that are perceived as most difficult. This software design technique is targeted for hard real-time embedded applications where the most difficult aspect of the project is meeting the timing requirements. In these applications, the correct functioning of the system depends on proper timing as much as the correctness of the

calculations. A key component of the method is to prototype those algorithms that are known to be required in the system, but their execution time is difficult to accurately estimate. The resulting prototype data is used to develop timing budgets and design a software structure to insure correct timing. Emphasis is placed on meeting software deadlines first, and to get the exact functionality later. However, functionality must be sufficiently correct to model the timing accurately. This approach is driven by previous experience that it is often much easier to "fix" the functionality rather than the performance. Put another way, it can be extremely difficult to improve the performance of a system that is grossly out of specification. Systems that are initially five to twenty times too slow are not uncommon and frequently result in complete redesign. This places timing on top of the "risk" list. When the functionality of some processing is not well understood, these are also appropriate areas to prototype. Prototypes may be utilized in the final product providing they are upgraded to insure compliance with coding styles. The process associated with the design method used is described as the Ada Time Oriented Method (ATOM). By knowing the difficulty of the real-time aspect, the program can be written to maximize maintainability while still meeting the performance objectives. The general philosophy is to always have a spectrum of choices to select from that provide increasing performance, at the sacrifice of memory, complexity, and ease of maintenance. A program that is maintainable but does not function because of performance problems is just as useless as a program that functions but is not maintainable. Both objectives are essential.

The implemented application code consisted of 3,200 lines of Ada and 31 code statements. There were 31 library units/subunits containing eleven tasks, one of which was an interrupt handler. The runtime code to support distribution was 900 lines of assembly language. This was done to be compatible with the vendor supplied runtime which was implemented entirely in assembly language. The top level design of the BDS is shown in Figure 1.

PROBLEMS AND SOLUTIONS

The most crucial demand for real-time performance came from rocket guidance requirements. To achieve the accuracy necessary to correct for flight trajectory errors and target acceleration, each of twenty rockets had to be

provided new guidance aimpoints every 100ms. Fixed point calculations were used to improve processing throughput. Previous experience had provided warning that it was difficult to get fixed point numbers with a 'SMALL' that is not a power of two (most implementations restrict representation clauses on 'small' to a power of two). Therefore, it was imposed on the system design that the hardware provided all dimensions as a power of two value.

Another task that was throughput intensive was the graphics display update. Although much less computation oriented than rocket calculations, the sheer volume of transactions made real-time response difficult. With 100 targets, 20 rockets, a

reticle, and statistic information each being revised, as many as 1,235 screen updates per second were required. An update consists of erasing and redrawing a symbol containing between six and thirty-three pixels. This provided an average pixel write time of 20,600 pixels/second or 48 μ s/pixel. This was the only place where inline code statements were utilized for performance. These statements provided variable shift operations that could not be achieved with the Ada code generator.

Finally, the operator pointing device imposed the most severe interrupt response-time requirement. To achieve a system specification of a 50ms update rate, the hardware had to be configured to respond every 28ms with a five byte data stream

BDS Top Level Design

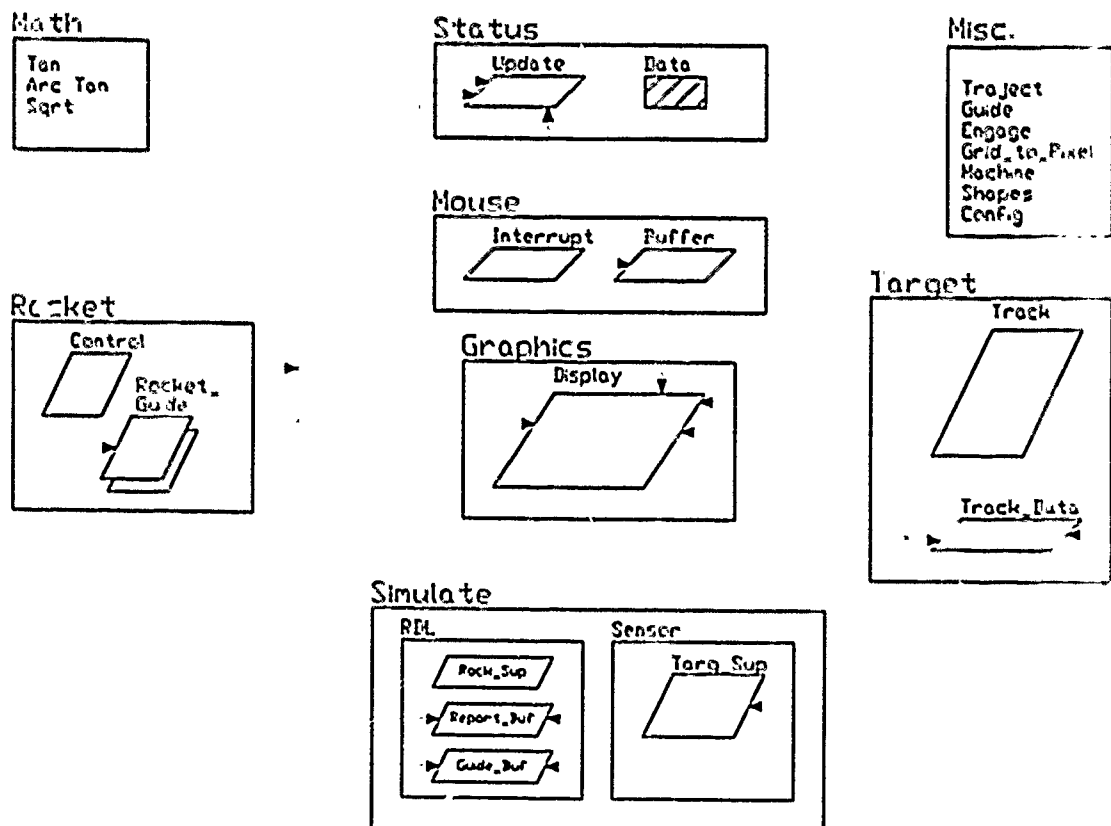


Figure 1 - BDS Top Level Design

at a rate of 2ms/byte. The solution was to use an implementation dependent interrupt handler pragma to execute the interrupt code without a full task context switch. As data from the pointing device was collected, a small amount of processing was performed in the interrupt routine and the time consuming functions were off-loaded to a background buffer task.

Some of the major problems encountered were because the "extended" features of Ada are not widely used and they have the greatest number of anomalies. The notable ones are as follows:

1) LONG_FIXED division was unreliable. Certain numbers (resulting in bit patterns very close to 1FFFFn) caused divide error. This was manifested by causing a NUMERIC_ERROR after hours of operation and hundreds of rocket launches and target intercepts. It was solved by using an exception block which altered the expression slightly and recomputed the value;

2) There was improper inlining of code statements. If the last instruction of the calling sequence used the same register as the first instruction of the machine code procedure to be inlined, the code generator would exchange the two instructions, for example,

```
mov [bp-10], ex
mov ex, [bp-20] --code statement begin
```

...
would become

```
mov ex, [bp-20]
mov [bp-10], ex --reordered, results in storing
--incorrect value.
```

Note that this problem appeared after the code in question had already undergone successful integration testing, i.e. after a re-compilation caused different registers to be used (with no changes in compiler switches). The solution was to use "mov ex, ex" which can be re-ordered with no effect;

3) Complex expressions did not always generate the correct code sequence. Actual parameters containing array aggregates, which in turn consisted of multidimensional array references with non-integer subscripts, resulted in a failure for the appropriate segment register to be loaded correctly. The graphics task takes a parameter list consisting of the old and new positions of an object (x,y), the object type (rocket, target, etc.), and a color. To determine the color, an array indexed by the object type was used in one dimension and a status flag indicating if it was engaged for intercept was used in the other dimension. This causes targets to "light up" when engaged for intercept. However, it did not work and the code was rewritten to create temporaries during each step of the

expression. The failure mode was to select the zero color, black, which gave the appearance that nothing was working, when in fact invisible targets were moving on the screen;

4) The pragma to establish task storage size did not function. This resulted in the program terminating before initial elaboration was complete. The program would simply crash with no exception trace back due to the fact that the program had not completed elaboration. This required single-stepping through the code to locate the problem. The solution was to use a linker option to set the library stack size, although it then applied the same stack size for all library tasks; and

5) Package CALENDAR elaboration check was not performed properly. A number of problems with elaboration were encountered due to library tasks starting execution before other units were elaborated. One strange problem was caused because no elaboration check was performed prior to calling the CALENDAR.CLOCK function to establish the periodic start point. Apparently the CALENDAR package body had not been elaborated and the CLOCK function returned the time of a few hundred microseconds of mission time. Then after the calling task was suspended for a rendezvous the CALENDAR package was elaborated, which set the TIME value to some time in 1987 (a very large number). When the task resumed execution, it reached the end of its loop and attempted to compute the delay necessary to achieve the desired interval. Since the delta time was almost 2,000 years, it exceeded the range of duration and a NUMERIC_ERROR was raised (although a TIME_ERROR should have been raised).

Finally, a problem with the Ada language surfaced. There are no provisions to perform a sequence of application statements and a runtime service such as ACCEPT without the possibility of intervening preemption. The application has a requirement to accept frequent interrupts, buffer the data to a certain point (based on the input stream), and then perform a considerable amount of processing on the data, while new data is arriving. This is done by having an interrupt task perform the buffering, then passing the data off to a background task. The problem is that the interrupt task may not be suspended for any reason other than to service higher priority hardware interrupts. This implies that a conditional rendezvous is required. However, what is really required is the ability to queue the buffer and request, then signal the background task if it is suspended waiting for new data. Essentially there are two approaches to handling this problem: 1)

provide a sufficient number of buffer tasks so that they can act as surrogates on the entry queue of the background task, or 2) maintain a flag that is only set when the buffer task is ready to immediately accept a rendezvous. This requires that the background task disable any type of preemption, check if there is more work to do, and if not, perform the accept statement. Presumably the runtime will then allow preemption only after placing the background task in a position to immediately accept the rendezvous. The interrupt task obviously will not attempt a rendezvous with the background task unless the flag is set. Both of these solutions have serious drawbacks. The surrogate task approach requires substantial optimization on the part of the compiler and runtime. Furthermore, it may make it less clear about the intent of the various rendezvous. The second approach is very implementation dependent, and is prone to error if used by other than very experienced and careful programmers. What is clearly needed is a simple asynchronous form of task communication. Perhaps a standard pragma designating a task as a surrogate, in which a call to its entry is guaranteed to have the same effect as a "signal" to the third party task would be a solution. To depend on implementation optimizations for such a crucial real-time operation is a poor approach to language design.

PRINCIPLE FINDINGS

Some of the principle findings of this project are as follows:

1) Although Ada compilers are near to being "full" implementations of the language, some of the most complex features may not be sufficiently reliable for life-critical applications. Several errors in runtime code have been detected under special operating conditions. These conditions include essentially random coincidence of executing groups of instructions while an external event invokes a context switch. This type of error may go undetected after years of operation, only to result in total system failure at a particular instant;

2) The execution rate of both generated code and the runtime code is considerably better than that of compilers of 1986. However, checking code remains verbose. This will tempt real-time application developers to suppress the checks, which has a consequence of taking different paths through the code generator. Since these paths may not have been tested as thoroughly as the primary path, the resulting code could be less reliable.

3) Design for distribution must have some

initial consideration, but does not require detailed information regarding the configuration of the target hardware, i.e. the number of processors. Limiting the amount of shared data is a general objective to facilitate distribution. To fully utilize all available processors, a design should implement independent activities of reasonable size as tasks rather than as procedures. If one complex sequence of calculations is not dependent on a previous set, it potentially could be done on more than one processor. "Reasonable" must be defined as a function of the overhead associated with a rendezvous as compared with a procedure call, weighed against the execution time of the activity.

4) A software manager should not use Ada on a serious real-time project without source code to the runtime. This is not for the purposes of modifying it, but to understand its detailed execution when necessary. This information is not available even in the best vendor documentation (which is often incorrect anyway) and can only be verified by examining the source of the runtime;

5) The Ada rendezvous model is practical, although not necessarily ideal, for distributed communication. Unconditional rendezvous with small parameter lists can be achieved with off-the-shelf communication hardware in under 1ms. Although this is significantly higher than the 100ns required for local rendezvous, it is still acceptable for many applications. More complex rendezvous mechanisms such as timed entry calls and selective waits with delay alternatives impose substantial additional overhead. As with non-distributed applications, the synchronous nature of the Ada rendezvous imposes additional task constructs in order to "uncouple" many inter-task communications;

6) In programs using tasks, default values for task stack size and task priority, as well as compiler selected elaboration order are unlikely to be suitable for most applications. Instead, designers should explicitly specify values for all task priorities and storage-size. Also the appropriate elaboration order must be conveyed to the compiler via the Elaborate pragma;

7) The impact of having many failures in the runtime and generated code is demoralizing to the engineering staff. It becomes apparent that the most difficult problems to find are those of the runtime and generated code, since one expects the Ada to work as specified. No matter how good the developers are, the system will not work if it won't do what it is instructed to by the Ada source code. This is unusual for real-time programmers who are familiar with assembly language where there are far fewer

discrepancies between the source and generated code;

8) The speed improvement of distributed Ada is not necessarily scalable. Although the parallel nature of embedded applications make them ideal for multiple processors, the individual tasks are not usually balanced in processor loading. On a shared memory multi-processor, scheduling can occur on a "next available processor" basis but this is usually not practical on a distributed system due to the locality of data. The "vectorized task" is a partial solution to this problem. To implement the guidance operation for up to twenty simultaneous rocket trajectories, an array of tasks was used. The actual size of the array was controlled by a configuration parameter. Each task in the array was passed a list of rockets to guide. If additional processors become available the size of the array can increase and the tasks can be distributed. The size of the individual "work" lists for each task would decrease correspondingly. This achieves a "near scalable" performance increase as processors are added;

9) Achieving distributed Ada via pre-processing the source code, or post-processing the generated code/runtime is acceptable for research, but unlikely to be usable for a production environment. What is really needed is an integrated compiler/linker/tester that supports distribution. An ideal compilation system would support a hybrid approach of distribution, i.e. clusters of shared-memory multiprocessors connected by a network;

10) Some aspects of the Ada language definition are silent about what should happen in a distributed system. For example, if a node fails, should future rendezvous to a task in that node get TASKING_ERROR or simply deadlock? What about a rendezvous already in progress with a failed node? What if the node fails, but then returns to service? These are all likely scenarios in typical distributed systems. Another area is the interpretation of the timed entry call. If the delay duration is greater than 0.0 and yet the delay expires prior to a message being sent to the remote task, should the rendezvous be terminated even if the accepting task is ready for an "immediate" rendezvous? A clear statement about what can be expected in these situations (or possibly control over what happens via pragmas) is necessary in future language revisions. Although many of these have been identified previously, no resolutions have been adopted and it is hoped that this work will shed some insight into how they may be resolved in future interpretations/revisions of the language;

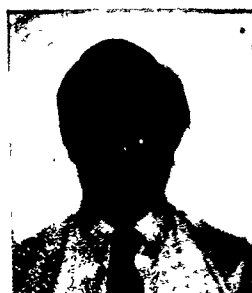
CONCLUSIONS

The latest release of Ada compilers are now supporting the features required for real-time embedded applications. Performance of Ada tasking operations is better than an order of magnitude over compilers of just a few years ago and optimizations, such as the execution of interrupt tasks without the cost of a full task switch, are now available with very good execution performance. As with any new software product, these new features must be used with special attention to insure that they perform as expected. Users should anticipate that these features may be less reliable as compared to more tested features, until they have received the usage necessary to work out small anomalies.

The use of Ada tasking constructs for distributed processing extends the benefits of compiler checking and a uniform model of concurrency beyond individual processors. Flexibility is enhanced since migration of function from one processor to another is now restricted only by communication requirements, which are being reduced substantially by the next generation of fiber-optic data links. Using distributed Ada to help relieve the processing requirements on a single processor appears to be a viable solution for many real-time applications.



Mrs. Mary E. Bender is a computer scientist with the Center for Software Engineering, U.S. Army CECOM, Ft. Monmouth, NJ. She is the project leader for their technology program in Ada real-time applications and runtime environments. She received a B. A. degree in Computer Science from Rutgers University, New Brunswick, NJ.



Mr. Thomas E. Griest is president of LabTek Corporation, Woodbridge, CT and engineering manager on LabTek's real-time distributed Ada project. He has been a principal member of the SIGAda Ada Runtime Environment Working Group since its inception, and serves as leader of the Implementation Dependencies Subgroup as well as a member of the Distributed Ada Task Force. Mr. Griest received his bachelor's degree in Computer Science from the State University of New York, College at Oswego.

MODIFICATION OF LU FACTORIZATION ALGORITHM FOR PARALLEL PROCESSING USING TASKS SUPPORTED BY ADA LANGUAGE

Shantilal N. Shah

Norfolk State University, Norfolk, VA.

An Algorithm to factor a given nonsingular matrix A into two Lower and Upper triangular matrices is modified so that the Lower triangular matrix can be computed by one task and the Upper triangular matrix can be computed by a second task, with both tasks running in parallel.

Existing Serial Algorithm:

To transform an NxN nonsingular matrix A into the product of two matrices L and U, where L is a lower triangular matrix and U is an upper triangular matrix with 1's on its main diagonal, the algorithm used is as follows:

```
DO FOR I = 1 to N
  L(I,1) = A(I,1)
END DO(I).
DO FOR J = 1 to N
  U(1,J) = A(1,J) / L(1,1)
END DO(J).
DO FOR I = 2 to N
  DO FOR J = I to N
    DO FOR K = 1 to (I-1)
      accumulate the SUM of
        L(J,K) * U(K,I)
    END DO(K).
    L(J,I) = A(J,I) - SUM
  END DO(J).
  U(I,I) = 1.
  DO FOR J = (I+1) to N
    DO FOR K = 1 to (I-1)
      accumulate the SUM of
        L(I,K) * U(K,J)
    END DO(K).
    U(I,J) = [A(I,J) - SUM] / L(I,I)
  END DO(J).
END DO(I).
```

In this algorithm, at each step I, where I = 1 .. (N-1), a column of L, the lower triangular matrix is computed first followed by a row of U, the upper triangular matrix in a serial mode. This process is continued until all the columns of the lower triangular matrix and all the rows of the upper triangular

matrix are computed.

At the Ith step, to compute the Ith column of L, the algorithm requires the elements L(J,K), where J = I .. N, K = 1 .. (I-1); and the elements U(K,I), where K = 1 .. (I-1). To compute the Ith row of U, the algorithm requires the elements L(I,K), where K = 1 .. (I-1); and the elements U(K,J), where K = 1 .. (I-1); J = (I+1) .. N.

For example, if A is a 5x5 matrix and we are at the stage to compute the third column of L and the third row of U, the computation of the third column of L is as follows:

$$L(3,3) = A(3,3) - [L(3,1) * U(1,3) + L(3,2) * U(2,3)]$$

$$L(4,3) = A(4,3) - [L(4,1) * U(1,3) + L(4,2) * U(2,3)]$$

$$L(5,3) = A(5,3) - [L(5,1) * U(1,3) + L(5,2) * U(2,3)]$$

The computation of the third row of U is as follows:

$$U(3,4) = [A(3,4) - [L(3,1) * U(1,4) + L(3,2) * U(2,4)]] / L(3,3)$$

$$U(3,5) = [A(3,5) - [L(3,1) * U(1,5) + L(3,2) * U(2,5)]] / L(3,3)$$

Requirements for a Parallel Algorithm:

The analysis of the above computations shows that the computation of the third column of L uses the elements U(1,3) of the first row of U and U(2,3) of the second row of U. It does not require any element of the third row of U. Similarly, the computation of the third row of U uses L(3,1) of the first column of L, L(3,2) of the second column of L and L(3,3) of the third column of L. If L is computed by one task, (task LOWER), and U is computed by another task, (task UPPER), in parallel, then at the third step, task LOWER, computing the third

column of L does not require any information from the task UPPER, computing the third row of U, provided that task UPPER has communicated the results of its computations of the first and second rows to task LOWER before it begins computations for the third row of U. Task UPPER does, however, require the value of $L(3,3)$ from task LOWER, computing the third column of L. This is the first and only element whose value is needed by task UPPER for the computation of the third row of U from task LOWER.

There are two means by which this requirement may be facilitated:

(1) After task LOWER has completed the computation of $L(3,3)$, it may communicate the value of $L(3,3)$ to task UPPER and then compute the rest of the third column.

(2) Task UPPER computes the value of $L(3,3)$ by itself and does not wait on task LOWER to communicate to it.

It is assumed here that each task, UPPER and LOWER, communicates the results of its row and column computations, to the other task, before it starts on the next row or column. Either of these alternatives will then allow task LOWER to compute the third column of L and task UPPER to compute the third row of U in parallel. The second alternative is the more desirable one for the following reasons:

(1) Task UPPER does not have to wait for task LOWER to communicate this result.

(2) It reduces the communications between the two tasks.

(3) Task LOWER can skip the computation of $L(3,3)$, because it does not need it for the computation of the third column of L.

This situation is true for every I th column of L and I th row of U, where $I = 2 \dots (N-1)$.

At the I th step, where $I = 2 \dots (N-1)$, task LOWER has to compute $(N-I)$ elements of the I th column of L and task UPPER has to compute $(N-I)$ elements of the I th row of U. Computations of these elements do not depend on each other and it can also be accomplished in parallel using several tasks by each LOWER and UPPER tasks. Communications, between the LOWER and UPPER tasks, required to pass the results of the computations, can be further reduced if the factors L and U of A are stored in matrix A (in place)

rather than in separate matrices L and U. This will require matrix A to be declared as a global variable (shared variable) for the LOWER and UPPER tasks. This will eliminate the need for communications between the two tasks as the computations of each row and each column are available. Each task will still be required to communicate with the other task to indicate that it has completed computation of the column or row that it was working on before it starts to compute the next column or row. Numerical experiments to compare the CPU time of these various approaches with different size matrices are in progress. The results of these studies should be available by the next joint Ada meeting. The modified algorithm for LU FACTORIZATION, using in place storage of L and U, and its implementation in Ada language are listed below.

Modified LU FACTORIZATION Algorithm :

Begin

Task LOWER:

1. Accept N, the size of matrix A.
2. Create tasks $X(I)$, $I = 1 \dots (N-2)$ to compute the elements of a column of the lower triangular matrix.
3. Compute $U(1,2)$, the second element of the first row of U and store it in $A(1,2)$.
 $A(1,2) = A(1,2) / A(1,1)$
4. Compute the columns of the lower triangular matrix
 DO FOR $I = 2 \dots (N-1)$
 - 4.1 DO FOR $J = (I+1) \dots N$
 CALL task $X(J-2)$ to compute the $L(J,I)$ element of L.
 END DO(J).
 - 4.2 DO FOR $J = (I+1) \dots N$
 CALL task $X(J-2)$ to check whether it has completed the computation of the element $L(J,I)$.
 END DO(J).
 - 4.3 Rendezvous with task UPPER to check whether it has completed the computation of row I.
 - 4.4 End the task $X(I-1)$ which is no longer needed.
 END DO(I).
5. Compute $A(N,N)$.


```

DO FOR K = 1 .. (N-1)
  accumulate the SUM of
    A(N,K) * A(K,N)
END DO(K).
A(N,N) = A(N,N) - SUM.

```

6. Communicate matrix A to the calling program.

END task LOWER.

Task UPPER:

1. Accept N, the size of matrix A.

2. Create tasks Y(I), I = 1 .. (N-2) to compute elements of a row of upper triangular matrix.

3. Compute the first row of U and store it in the first row of A.
A(1, 3..N) = A(1, 3..N) / A(1,1)

4. Compute the rows of the upper triangular matrix

```
DO FOR I = 2 .. (N-1)
```

4.1 Compute L(I,I), the Ith row and the Ith column element of L, needed for computation of the Ith row of U. Store it in A(I,I).

```

DO FOR K = 1 .. (I-1)
  accumulate the SUM of
    A(I,K) * A(K,I)
END DO(K).
A(I,I) = A(I,I) - SUM.

```

4.2 DO FOR J = (I+1) .. N
CALL task Y(J-2) to compute the U(I,J) element of U.
END DO(J).

4.3 DO FOR J = (I+1) .. N
CALL task Y(J-2) to check whether it has completed the computation of element U(I,J).
END DO(J).

4.4 Rendezvous with task LOWER to check whether it has completed the computation of the column I.

4.5 End the task Y(I-1) which is no longer needed.

```
END DO(I).
```

END task UPPER

Tasks X(I) and Y(I), I = 1 .. (N-2):

1. Accept the indices J and K of the element to be computed and the index L_OR_U for LOWER or UPPER matrix.

2. Compute the element L(J,K) or U(J,K) and store it in A(J,K).

```

2.1 If element of L (L_OR_U=0) then
  DO FOR K1 = 1 .. (K-1)
    accumulate the SUM of
      A(J,K1) * A(K1,K)
  END DO(K1).
  A(J,K) = A(J,K) - SUM.

```

else

```

(L_OR_U=1)
DO FOR K1 = 1 .. (J-1)
  accumulate the SUM of
    A(J,K1) * A(K1,K)
END DO(K1).
A(J,K) = 1A(J,K) - SUM / A(J,J)

```

end if.

3. Accept the index of the task X or task Y to check completion of the computation.

4. accept index I to stop task X(I) or Y(I).

END task X(I) or Y(I).

Ada Code:

generic

BOUND : in INTEGER;
package LU_DECOMP_ROUTINESLL is

```

type MATRIX is
array(1 .. BOUND, 1 .. BOUND) of FLOAT;
procedure LU_DECOMP(A : in out MATRIX;
  BOUND : in INTEGER);

```

end LU_DECOMP_ROUTINESLL;

package body LU_DECOMP_ROUTINESLL is

```

-- Procedure to factor a given matrix
procedure LU_DECOMP(A : in out MATRIX;
  BOUND : in INTEGER) is

```

```

LU_DONE : BOOLEAN := FALSE;
SINGULAR : exception;

```

```

-- procedure to compute the sum of
-- A(INDX1,K) * A(K,INDX2)

```

```

-- for row and column elements
procedure SUM(
  INDX1, INDX2 LIMIT : in INTEGER;
  S : in out FLOAT);

-- task to compute LOWER TRIANGULAR
-- FACTOR

task LOWER is
  entry N_IN_LOWER(
    BOUND : in INTEGER);

  entry STOP_LOWER(
    LU_DONE : in BOOLEAN);
end LOWER;

-----

-- task to compute UPPER TRIANGULAR
-- FACTOR

task UPPER is
  entry N_IN_UPPER(
    BOUND : in INTEGER);

  entry ROW_COLUMN_CHECK(
    COLUMN_DONE : out BOOLEAN);

  entry STOP_UPPER(
    LU_DONE : in BOOLEAN);
end UPPER;

-----

-- task to compute an element of
-- row or column of factor matrix

task type COMPUTE is
  entry ELEM( FIRST, SECOND,
    CHOICE : in INTEGER);

  entry ELEM_DONE( DONE1
    : out INTEGER);

  entry STOP_TASK( DONE
    : in INTEGER);
end COMPUTE;

-----

-- body of the procedure SUM
procedure SUM( INDX1, INDX2,
  LIMIT : in INTEGER;
  S : in out FLOAT) is
begin
  S := 0.0;
  for K in 1 .. LIMIT loop
    S := S + A(INDX1, K)*A(K, INDX2);
  end loop;
end SUM;

-----

-- body of the task LOWER
task body LOWER is
  N, CHOICE : INTEGER;
  N_IN_L : BOOLEAN := FALSE;

```

```

COLUMN_DONE : BOOLEAN := FALSE;
NEXT_COLUMN : BOOLEAN := FALSE;
L_DONE : BOOLEAN := FALSE;
S : FLOAT := 0.0;

begin
  -- get index to compute column element
  L_OR_U := 0;
  loop
    select
      -- get bound in LOWER
      accept N_IN_LOWER(
        BOUND : in INTEGER) do
        N := BOUND;
        end N_IN_LOWER;

      -- compute second element of first
      -- row of U and store it in A(1,2)
      A(1, 2) := A(1, 2)/A(1, 1);
      N_IN_L := TRUE;

    or
      delay 0.1;

    end select;
    exit when N_IN_L;
  end loop;

  declare

    -- tasks to compute elements of
    -- column of L
    X : array(1..(N - 2)) of COMPUTE;
    X_DONE: array(1..(N - 2)) of INTEGER
      := (1 .. (N - 2) => 0);

  begin
    for I in 2 .. (N - 1) loop

      -- begin computations of elements
      -- of Ith column of L using tasks
      for J in (I + 1) .. N loop
        X(J-2).ELEM(J, I, L_OR_U);
      end loop;

      -- check all tasks X are done
      -- with Ith column of L
      for J in (I-1) .. N loop
        X(J-2).ELEM_DONE(X_DONE(J-2));
      end loop;

      -- rendezvous with task UPPER to
      -- check whether it has finished
      -- computation of Ith row
      UPPER.ROW_COLUMN_CHECK(
        COLUMN_DONE);

      -- stop task X no longer needed
      X_DONE(I-1) := 1;
      X(I-1).STOP_TASK(X_DONE(I-1));

      -- next column of L
    end loop;

    -- last column of L
    SUM( N, N, N - 1, S);

```

```

A(N, N) := A(N, N) - S;
L_DONE := TRUE;

-- terminate task LOWER
loop
  select
    when L_DONE =>

      accept STOP_LOWER(
        LU_DONE : in BOOLEAN) do
        L_DONE := LU_DONE;
      end STOP_LOWER;
    exit when L_DONE;

  or

    delay 0.1;
  end select;
end loop;
end;
end LOWER;

```

task body UPPER is

```

N, L_OR_U      : INTEGER;
ROW_COLUMN_DONE : BOOLEAN := FALSE;
N_IN_U         : BOOLEAN := FALSE;
ROW_DONE       : BOOLEAN := FALSE;
U_DONE         : BOOLEAN := FALSE;
S              : FLOAT := 0.0;

```

begin

```

-- index to compute row element
L_OR_U := 1;
loop
  select
    -- get bound of A
    accept N_IN_UPPER(
      BOUND : in INTEGER) do
      N := BOUND;
    end N_IN_UPPER;
    N_IN_U := TRUE;

    -- compute first row of U;
    if (A(1,1) = 0.0) then
      raise SINGULAR;
    end if;
    for J in 3 .. N loop
      A(1, J) := A(1, J)/A(1, 1);
    end loop;

```

or

```

  delay 0.1;

```

```

end select;

```

```

exit when N_IN_U;

```

```

end loop;

```

declare

```

-- tanks to compute row elements of U
Y : array(1 .. (N - 2)) of COMPUTE;
Y_DONE : array(1 .. (N - 2)) of INTEGER
       := (1 .. (N - 2) <> 0);
begin

```

```

  for I in 2 .. (N - 1) loop
    -- compute the first element of
    -- Ith column of L
    SUM(I, I, I - 1, S);
    A(I, 1) := A(I, 1) - S;

```

```

    -- begin computation of Ith row
    -- of U using tanks
    for J in (I + 1) .. N loop
      Y(J - 2).ELEM(I, J, L_OR_U);
    end loop;

```

```

    -- check all tanks Y are done with
    -- Ith row of U
    for J in (I + 1) .. N loop
      Y(J - 2).ELEM_DONE(Y_DONE(J - 2));
    end loop;
    ROW_DONE := TRUE;

```

```

    -- stop task no longer needed for
    -- row computation
    Y_DONE(I - 1) := 1;
    Y(I - 1).STOP_TASK(Y_DONE(I - 1));

```

```

  loop
    select
      when (ROW_DONE or U_DONE) =>

```

```

        -- rendezvous with task LOWER to
        -- check for Ith row and column
        -- completed
        accept ROW_COLUMN_CHECK(
          COLUMN_DONE : out BOOLEAN) do
          COLUMN_DONE := ROW_DONE;
        end ROW_COLUMN_CHECK;
        U_DONE := FALSE;
        ROW_DONE := FALSE;
        ROW_COLUMN_DONE := TRUE;

```

or

```

    delay 0.1;
  end select;
  exit when ROW_COLUMN_DONE;
end loop;
ROW_COLUMN_DONE := FALSE;

```

```

-- next row of U
end loop;

```

```

-- all rows of U done
U_DONE := TRUE;

```

```

-- terminate task UPPER

```

```

loop
  select
    when U_DONE =>

```

```

    accept STOP_UPPER(
      LU_DONE : in BOOLEAN) do
      U_DONE := LU_DONE;
    end STOP_UPPER;
    exit when U_DONE;

```

```

    or
        delay 0.1;
    end select;
end loop;
end;
end UPPER;
-----
task body COMPUTE is
    INDX1, INDX2, LIMIT : INTEGER;
    DONEC                : INTEGER := 0;
    S                    : FLOAT   := 0.0;
    INDX                 : INTEGER ;
    BDONE                : BOOLEAN := FALSE;

begin
    loop
        select

            -- accept the indices of the element
            -- to be computed and index for
            -- the matrix
            accept ELEM(FIRST, SECOND, L_OR_U
                       : in INTEGER) do
                INDX1 := FIRST;
                INDX2 := SECOND;
                INDX  := L_OR_U;
            end ELEM;

            if (INDX = 0) then
                -- computing column element
                LIMIT := INDX2 - 1;
                SUM(INDX1, INDX2, LIMIT, S);
                A(INDX1, INDX2) :=
                    A(INDX1, INDX2) - S;
                BDONE := TRUE;

            else

                -- computing row element
                LIMIT := INDX1 - 1;
                SUM(INDX1, INDX2, LIMIT, S);
                if (A(INDX1, INDX2) = 0.0) then
                    raise SINGULAR;
                end if;
                A(INDX1, INDX2) :=
                    (A(INDX1, INDX2) - S) /
                    A(INDX1, INDX1);
                BDONE := TRUE;

            end if;

        or

            when BDONE =>

                -- accept index for element done
                accept ELEM_DONE(
                    DONE1 : out INTEGER) do
                    DONE1 := DONEC;
                end ELEM_DONE;

            or

                -- accept index for task to be
                -- stopped
                accept STOP_TASK(
                    DONE : in INTEGER) do
                    DONEC := DONE;
                end STOP_TASK;

```

```

        exit when (DONEC = 1);
    or
        delay 0.1;
    end select;
end loop;
end COMPUTE;
-----
-- begin the LU_DECOMP procedure
begin

    -- pass the bound of A to tasks
    -- LOWER and UPPER
    LOWER.N_IN_LOWER(BOUND);
    UPPER.N_IN_UPPER(BOUND);

    -- terminate tasks LOWER and UPPER
    LU_DONE := TRUE;
    LOWER.STOP_LOWER(LU_DONE);
    UPPER.STOP_UPPER(LU_DONE);

    -- abort tasks if matrix is singular
    exception
        when SINGULAR =>
            abort UPPER, LOWER;
end LU_DECOMP;
end LU_DECOMP_ROUTINES;
-----

```

This research work has been sponsored by a research and equipment grant from the United States Army (ARO proposal #25510-EL-II) by way of the Army Research Office and AIRMICS.

About the Author

Shantilal N. Shah is a Professor of Computer Science in the Department of Mathematics and Computer Science at Norfolk State University. He received his Ph.D. in Mathematics from the Syracuse University in 1972 and his M.S. in Computer Science from the College of William and Mary in 1975. His current research interests include Parallel Processing and Parallel Algorithms.

Department of Mathematics and
Computer Science
Norfolk State University
2401, Corpview Avenue
Norfolk, Virginia 23504



Learning Ada From Ada

Lawrence E. Smithmier Jr.

University of Mississippi
Undergraduate Student Paper

On-line tutorials can provide an effective teaching tool within a computer environment. They can be designed in such a way as to accommodate users with different levels of knowledge and who work at different paces.

This paper discusses a tutorial written in Ada on an IBM 370 using the Alsys compiler. Our system, designed and written using Ada, allows the users to move forward or backward within the tutorial because three screens (the current screen, the next screen, and the previous screen) are kept in memory by background tasks. When the user proceeds to another screen, a background task brings the next appropriate screen into memory while the foreground task displays the requested screen. This facilitates a quicker response time when moving through the tutorial. A balance between maximum speed and minimum memory usage is achieved through this use of Ada tasking. Through the use of packages, the screen driver and the questionnaire driver can both use the same background task. The tutorial is written entirely in Ada using only one IBM system call, done via a call to low-level I/O an Ada library call supported in the LRM.

Ada is very similar to Pascal in the types of control structures available, the types of operators used, the use of subprograms, and most of the data types available. It does, however, have some subtleties which are not intuitively obvious, for example: it is important to look carefully at the way tasking is handled and how and when it should be used. It is also important to look at generics, private types, and variable passing.

The on-line tutorial has been used effectively for some time as a teaching tool in a computer environment because it gives the user a hands-on feel for the subject. This makes such a tutorial a natural tool to teach a new and difficult language because it will allow the user to determine which language features he can review and which he needs to study in-depth.

Our tutorial was written as part of a software engineering course using Ada as a tool.

Ada was chosen because it so readily supported software engineering principles. Software engineering goals realized by Ada are: abstraction, automation, information hiding, localized costs, portability, preservation of information, simplicity, and structure.

This tutorial was the final group project of the semester and combines the knowledge we learned of both Ada and software engineering in one package. The tutorial was written modularly to aid in the maintainability of the code itself. It is also portable because all but one command is standard Ada. The tutorial also uses a buffering package which provides it with a level of information hiding. And finally, the tutorial is abstract because the names of lesson and quiz files to be used are stored in a file which can be changed to add or delete lessons.

The tutorial was written in Ada about Ada by students who were learning Ada as they were completing the assignment. The lessons and quizzes were also written by students who were learning as they tried to teach others. This makes the tutorial a much better teaching tool because the writers still remembered what was hard for them to learn and took the time to present these topics in a little more depth and from several angles. A good example of this is in the lesson tasking in which two program outlines are discussed. One of the examples is a program which readily lends itself to tasking: (Barnes, p. 209)

```
procedure SHOPPING is
  task GET_SALAD;

  task body GET_SALAD is
    begin
      BUY_SALAD;
    end GET_SALAD;
```

```

task GET_WINE;

task body GET_WINE is
begin
    BUY_WINE;
end GET_WINE;

begin
    BUY_MEAT;
end SHOPPING;

```

where BUY_WINE, BUY_SALAD & BUY_MEAT are procedures and GET_WINE & GET_SALAD are tasks. The other example shows a program which does not use tasking efficiently:

```

procedure FIX_A_FLAT is
    task TAKE_OFF_TIRE;

    task body TAKE_OFF_TIRE is
    begin
        REMOVE_LUG_NUTS;
    end TAKE_OFF_TIRE;

    task PUT_ON_SPARE;

    task body PUT_ON_SPARE is
    begin
        REPLACE_LUG_NUTS;
    end PUT_ON_SPARE;

begin
    GET_SPARE;
end FIX_A_FLAT;

```

where REPLACE_LUG_NUTS, REMOVE_LUG_NUTS & GET_SPARE are procedures and TAKE_OFF_TIRE & PUT_ON_SPARE are tasks.

The tutorial allows the beginner to proceed at his own pace and to check his mastery at the end of each topic using the quizzes. The quizzes are written as a series of question and answer sessions of varying length. The questions were designed to test the user on the concepts we felt were essential that the user know for effective programming. The quizzes are provided at the end of each lesson and the user has the option of taking them or skipping them. They are displayed by a separate process from the tutorial text. For example, the tasking quiz presents a problem definition and asks whether tasking should be used.

2. Should tasking be used in a program which will simulate a cross-country trip by car, assuming you don't need to eat? (no, because the operations left: driving,

sleeping, and getting gas, can not be done in parallel)

If the user gets a question wrong he is told the right answer and is given the total number missed upon completing the quiz.

The more sophisticated user will be able to skip topics he is well versed in because the tutorial is menu driven and the user need not review a subject unless he wants to, and even then he is given the option to quit that lesson. The option menu looks like this:

Ada Tutorial

- 1) OPERATORS
- 2) TYPES
- 3) CONTROL STRUCTURES
- 4) SUBPROGRAMS
- 5) TASKING
- 6) GENERICS
- 7) LIBRARY UNITS
- 8) EXCEPTIONS

Please enter the number of the lesson you would like to run or a 0 to exit:

This menu allows the user to not only choose the topics he will study, but also the order in which they will be studied. The order in which they appear on the menu does represent the order which we felt they should be covered. The files which are displayed on the menu come from a file which holds the name of each text file and quiz to be offered. The form of the file is:

```

OPERATORS
OPERATOR
TYPES
TYPE1
CONTROL STRUCTURES
CONTROL
SUBPROGRAMS
SPROGRAM
TASKING
TASKING
GENERICS
GENERIC
LIBRARY UNITS
LIBUS
EXCEPTIONS
EXCEPT
EXIT
NOFILE

```

where OPERATORS is the text file and OPERATOR is the questionnaire file. The control word NOFILE is used when no questionnaire is available and the code word EXIT is used to

indicate the end of the topic list. This system allows the tutorial program to be easily updated or even changed to a different subject entirely. This can be done by simply updating the file M_FILE to represent the current state of the tutorial.

The lessons are displayed by a screen handler which receives the information in an unformatted state from a text file. This increases the ease in which lessons can be corrected or replaced. The screen handler allows the user to move along at a quick pace over subjects he is somewhat familiar with or to go slowly through those subjects which he has not seen before by allowing the user to decide when the next screen is displayed. It also gives the user the option of backing up to the previous screen, or even quitting that lesson entirely. The lessons in the tutorial are taken from Programming in Ada Second Edition, by J. G. P. Barnes, and are listed roughly in the same order as found in the book. The lesson screen looks like this: (Barnes, p. 195, 4)

```
generic
  type ITEM is private;
  procedure EXCHANGE( X, Y: in out ITEM);
  procedure EXCHANGE( X, Y: in out ITEM) is
    T: ITEM;
  begin
    T:= X; X:= Y; Y:= T;
  end;
```

The subprogram EXCHANGE is a generic subprogram and acts as a kind of template. The generic mechanism takes the form of subprogram specification which is preceded by the generic formal part consisting of the reserved word "generic" followed by a (possibly empty) list of generic formal parameters. Note that we have to give both the body and the specifications separately.

The generic procedure cannot be called directly but from it we can create an actual procedure by a mechanism known as generic instantiation. For example, we may write

Enter an N, a P, or an E to move forwards, backwards, or to exit:

The tutorial is able to allow the user to back up through the use of a double buffering system. One buffer holds the previous screen, the other holds the next screen, while the current screen is being displayed. Both the questionnaire and the lesson procedures use the same buffering system. This is facilitated through the use of packages. The buffering package maintains two twenty-one line buffers at all

times, while a third buffer is held in the procedure currently running.

The tutorial uses two background tasks to update the buffers, thus speeding up screen updating and allowing the user to move both forward and backward in the text. One task holds the previous screen while the other holds the next screen. Each task reads from the tutorial text file and holds one page in memory at a time. The text in the files is unformatted, that is, the text is saved in the files in the same way it will be shown on the screen. The buffering system is defined as a package, which is used by both the text and questionnaire drivers. Through the use of the package, we made the code more readable and more compact. This use of background tasks and packages reduces both the complexity of the code and the amount of code in memory during execution.

The tutorial is written entirely in Ada using only one IBM system call. The system call was done via a call to low-level I/O, an Ada library call, supported in the LRM. This system call is a CLRSCRN command, used to clear the screen, and is accomplished as follows:

```
EXECUTE_COMMAND("CLRSCRN");
```

This is the only command which is not fully portable to any Ada machine.

The tutorial was written and tested using the Alsys IBM 370 ADA compiler for VM/CMS, version 2.3. The compiler was run on the University of Mississippi's Amdahl 470/V8 running IBM VM/SP CMS Version 4.0. The Amdahl 470/V8 is architecturally equivalent to the IBM 3083JX.

In conclusion, we feel that a tutorial written for Ada can be a valuable teaching tool. We think that having students who had recently learned Ada write a tutorial on the subject was useful because, as stated earlier, they remembered what was difficult for them to learn. It might also have been good to have Ada programmers at several higher levels of knowledge participate in the tutorial design. They could have given insight into more advanced implementation strategies used in complicated applications. Ada made the tutorial easier to write as a group because it supports separate compilation which allowed the parts to be written separately and then combined. The use of tasking also provided quick screen display, while packages allowed for better abstraction within the code. Writing this tutorial in Ada

gave us a first hand experience in how the structure of the language encourages good software engineering methods and shows how Ada allows a design group to work in parallel.

Barnes, J. G. P. *Programming in Ada*, 2nd. ed.
Reading, MA: Addison-Wesley Publishers
-Limited, 1983.



Lawrence E. Smithmier, Jr.
P. O. Box 8317
University, Mississippi 38677

Larry is a senior in the School of Engineering at The University of Mississippi. He will graduate in August 1989 with a B.S. in Computer Engineering and a B.A. in Mathematics. Larry is currently employed as a part-time programmer for the National Center for Physical Accustics .

Problems in Using Ada as a Development Tool

Allison Juanita Mull

The University of Mississippi

ABSTRACT

The purpose of this paper is twofold: firstly, to take a look at how ADA was used to support and uphold the goals and principles of software engineering within the context of developing a major interactive tutorial program to teach ADA, and, secondly, to describe some of the pitfalls of using ADA as a development tool. The primary problem examined concerns the issue of portability as related to the implementation of a required function which could not be provided within the ADA environment. Consequently, this caused the tutorial to be non-portable.

(I) INTRODUCTION : BACKGROUND

Realizing that ADA was one of the most significant developments in programming languages and wanting to promote some degree of ADA literacy at the University of Mississippi, one of the software engineering classes developed a tutorial program on ADA and its programming environment. The tutorial, called ADA2, was developed as a team effort. It consists of a single package, consisting of three procedures. It is fully interactive and was written to run on the university's Amhdal 470 V/8 running IBM VM/SP release 5. The Amhdal 470 V/8 is architecturally equivalent to the IBM 370. The compiler used in developing this project was Telesoft's Telegen2 ADA development system for the 370 -- A370 version 1.0.

Briefly, the operation of this program is as follows. The primary procedure called procedure Menu is responsible for displaying the main menu which consists of the topics which can be chosen by the user during a particular session. This procedure also controls the flow of operations during a tutorial session. The selection of topics available through procedure Menu in ADA2 is as follows:

RESPONSE TOPIC OF INTEREST

RESPONSE	TOPIC OF INTEREST
0	Table of Contents
1	Introduction
2	Runtime Environment
3	Subprograms
4	Packages
5	Exceptions/Handlers
6	Generics
7	Tasks
8	Tasks (in brief)
9	Quit

The user enters a response and the file corresponding to this response is passed to procedure Topic. Procedure Topic then reads the information in this file and displays it on the screen, filling the screen with each display. The user may either quit at this point or continue the tutorial. When all of the information in the Topic file has been displayed, the user is queried as to whether or not he would like to test his knowledge by being asked a series questions concerning it. If the user responds yes, Procedure Topic calls Procedure Runexer which allows the user the opportunity to respond to a series of true and false questions about the material. Once the user has finished testing his knowledge, or if he determined not to do so, he is returned to the menu of selections (Procedure Menu) once again.

(II) A DISCUSSION OF HOW THE ADA LANGUAGE WAS USED WITHIN THE CONTEXT OF THE ADA2 PROJECT TO UPHOLD THE GOALS AND PRINCIPLES OF SOFTWARE ENGINEERING.

With the ADA2 project in mind it is important to determine the way in which ADA was used during the development of the project to uphold the four basic goals of contemporary software engineering practices. According to Booch, the four basic goals of software engineering are:

- (1) Modifiability,
- (2) Efficiency,
- (3) Reliability, and
- (4) Understandability.

Modifiability refers to the ability to control the impact of changes in a software product; whereas, efficiency refers to the optimality with which a computer system uses its available resources especially time and space. Reliability refers to the degree to which a system is failure free and its ability to degrade and recover gracefully. Finally, understandability refers to the degree to which various people who examine the project can comprehend it and the degree to which they can isolate the objects and operations in the solution. The principles which ADA was designed to support which lead to these goals are the following:

- (1) Abstraction,
- (2) Information Hiding,
- (3) Modularity,
- (4) Localization,
- (5) Uniformity,
- (6) Completeness, and
- (7) Confirmability.

The first principle, abstraction, refers to the extraction of essential details of a given level of representation. The second principle, information hiding, makes inaccessible certain details which are not necessary for the proper functioning of the rest of the system. Both abstraction and information hiding aid in the maintainability and understandability of software by reducing the amount of details a developer must know at a certain level of representation. In addition, the reliability of software is enhanced when, at each level of abstraction, only those operations which do not violate the logical view at that level are available. Modularity, which is the third principle, is the grouping of functions or operations into modules according to some criteria. Localization, the fourth principle, is a principle aiding in the creation of modules with loose coupling, that is, modules that are highly independent, and in the creation of modules having strong cohesion, a quality exhibited when all the inner elements of a module are closely related. Both modularity and localization support the software engineering goals of modifiability, reliability, and understandability. This is because if a system is structured well, the ability to understand any given module should be enhanced, and since design decisions have been localized, the effects of modification to a module or modules will be limited. Also, if code is modularized well, then interconnections among modules will be limited, thus serving to enhance

reliability. The fifth principle, uniformity, supports the software engineering goal of understandability by ensuring that all the modules use a consistent notation and do not have any unnecessary differences. The sixth principle, completeness, ensures that all the important elements in a module are present. The seventh principle, confirmability, refers to the ease with which a system can be tested to confirm whether or not it meets requirements. Both support the goals of reliability, efficiency, and modifiability by aiding in the development of correct solutions.

Within the context of the ADA2 project, the ADA language and environment assisted in realizing these principles in following ways. The principle of information hiding was realized in our project through the use of the three procedures mentioned earlier. Each procedure communicated to the other via a well defined procedural interface with only the essential information available for its use. In realizing the principles of abstraction as well as information hiding, the power of the language was not fully exploited as can be seen by examining the package specification, which appears below.

```
WITH TEXT_IO, SYSTEM;
USE TEXT_IO, SYSTEM;
PACKAGE ADA2 IS
```

```
    PROCEDURE MENU;
```

```
    PROCEDURE RUNEXER(FT:FILE_TYPE);
```

```
    PROCEDURE TOPIC( FT : FILE_TYPE);
```

```
END ADA2;
```

Appearing in the package specification are some unnecessary details about the tutorial mechanics. Specifically, since both procedure Topic and procedure Runexer are subordinate to procedure Menu (as has been previously described), and since it would be unnecessary for any user to know of their existence, these two procedures should have been embedded within procedure Menu so that from the specification level only the declaration for procedure Menu would be visible. In this way, the principles of abstraction and information hiding would have been better realized.

The principle of modularity was realized by grouping the tutorial package ADA2 into three separate procedures according to the following criteria:

(1) the operations to be performed in procedure Menu were to be only those associated with displaying the menu of available topics to the screen. This included preparing each file for its

possible selection by the user by performing an open on each. This also included querying the user as to which selection he would like, and then passing the information file corresponding to his selection to procedure Topic.

(2) The operations to be performed in procedure Topic were to be only those associated with displaying the information from the file to the screen, a screen full at a time, until the end of the file was reached. Once that had been accomplished, procedure Runexer was called if the user indicated that he wished to test his knowledge of the topic he had just viewed.

(3) The operations performed in procedure Runexer are simply those required to read in the test questions for the user and to notify him of the status each response. Runexer also provides a final test score for the user.

The modules are not as loosely coupled as might be hoped, but this is due in large part to the nature of the project. For example, both procedure Topic and procedure Runexer depend upon the functioning of procedure Menu to obtain and pass the user's topic choice. In addition, all of the modules exhibit a large degree of cohesiveness since the operations within them are directed toward achieving a common goal (i.e., displaying the menu of choices and obtaining the user's topic selection, etcetera). Furthermore, the modules are all uniform since they were all designed in a top down manner. They are all complete since they have within them all of the important elements necessary to perform the desired operations. This completeness can be confirmed by having a user operate the tutorial.

Finally, it can be seen that, of the seven basic design principles stated earlier, some are upheld to a greater degree than others. It can, therefore, be concluded that even a language developed with the idea of supporting the software engineering principles cannot ensure the perfect application of these principles. Though ADA has the power to support these goals, any language, even ADA, can only be as powerful as its users allow it to be. That is, writers of software must still be carefully aware of software engineering principles and violate them only with care and only when no other solution seems to be available.

(III) PROBLEMS ASSOCIATED WITH THE USE OF ADA.

Now that the way in which ADA served to support the software engineering goals through the ADA2 tutorial project has been discussed, it is appropriate to consider some of the pitfalls which the ADA2 project had to overcome as a result of using ADA as the development language. A necessary

function for the smooth operation of the tutorial was the ability to clear the screen so that information could be displayed a screen at a time. (It is important to realize that this project was being done in a mainframe environment with minimal support for external devices. There was no library support for clear screen.) Without this function, the alternatives were either to output the information line by line, which eventually results in a screen full of material that has already been read with one new line at a time appearing at the bottom or to use a series of new line's, a solution which could not be guaranteed to always completely clear the screen, especially since system information would be printed to it occasionally, thus throwing off the functioning of the rest of the output statements designed to print information to the screen. It was assumed that the new page subprogram provided within the ADA environment to page files with the mode of "out" could be used to page the screen, but this did not work. This presented a dilemma. After attempting and failing to bind and link an assembly language module with the purpose of clearing the screen to the tutorial it was discovered that Pragma Interface could be used to interface a system dependent assembly language procedure which could be called at different points in the code to clear the screen. The use of this assembly language procedure caused the program to be non-portable to installations not possessing this feature, a problem one would not have expected to encounter within the ADA environment. It should be pointed out, however, that since the arrival of the Alsys ADA compiler at the University of Mississippi and the development of subsequent versions of tutorials written in ADA, this problem has been solved.

Another problem which could be viewed as a pitfall from which the ADA2 project team suffered was having to deal with the large amounts of space and time it took to compile and execute the code each time this was necessary. Though student computer disk space allotments are generally small, they are usually sufficient for most projects developed using other languages, even when these projects require thousands of lines. However, even for a modestly sized ADA program of a few hundred lines, temporary disk space of 10 to 20 extra cylinders would have to be set up and programs had to be shifted back and forth from the temporary space to a student's permanent space between log on and log off since temporary space disappears upon log off. Also, if there was any load upon the system at all when recompiling and executing the ADA2 program, it was not unusual to have to wait from thirty minutes to an hour to receive the results. An equivalent sized Pascal program would have

compiled under the same system load in under two minutes.

There are several other factors which could have contributed to this time inefficiency, but anything approaching this amount of inefficiency is generally viewed as being unacceptable. Yet again, since the arrival of the Alslys compiler, it has been reported that this time inefficiency has disappeared and ADA programs on the order of a few hundred lines typically take only a few minutes, at most, to return results.

(IV) CONCLUSION.

In conclusion, since many of the problems experienced with the ADA language seem to have disappeared with the introduction of the Alslys ADA compiler, it leads one to think that these problems might be due, in some part, to the Telegen2 ADA compiler under which the ADA2 project team worked. Furthermore, because subsequent student teams have been very successful in areas where the writers of ADA2 failed it can be concluded that many of the difficulties which were thought to be Ada related were compiler related and not always due to limitations of the language itself. Even so, a project having experiences such as these would tend to cause the members of its project team to look much more critically at a language and certainly much more carefully at compilers. Also, as has been previously stated and as can be seen, though ADA has the power to support the goals of software engineering, it does not have the power to enforce them. Therefore, software engineers must not look to ADA for some sort of magical panacea, instead they should see it for what it is: a very good tool; albeit, still a tool, which is subject to the limitations of its users.

NOTES

¹Grady Booch, Software Engineering With ADA, The Benjamin/Cummings Publishing Company, Inc., copyright 1983, p. 29.

²Booch, pp. 29-31.

³Booch, p. 31.

⁴Booch, pp. 32-35.

REFERENCES

Booch, Grady. Software Engineering With ADA. The Benjamin/Cummings Publishing Company, Inc. Copyright 1983. PP. 29-35.



Allison Junita Mull is a Graduate Student in Computer Science at the University of Mississippi.
Ms Mull is due to receive a Master of Science in Engineering Science in December 1989.

AN Ada SYSTEM FOR THE PARALLEL EXECUTION OF FP PROGRAMS

Norman Graham

Oklahoma State University

Abstract

This paper describes an attempt to bring program correctness proofs to the Ada environment by providing a FP to Ada translator and a run-time system for FP written in Ada. The FP run-time system uses Ada tasks to take advantage of the small grain parallelism inherent in FP programs. A tree of tasks is generated dynamically to compute the FP function in a manner that is based on graph reduction. An unexpected discovery is that Ada is unable to kill branches in this task tree as it is generated currently. To circumvent this problem, the run-time system is prevented from generating branches in the task tree until the branch is known to be required.

Motivations

Ada's traditional scope of applicability includes real-time systems, scientific programming, database systems, and distributed systems. To ensure the correct building of these systems, Ada supports and promotes the use of software engineering techniques. These techniques, when used with a referentially opaque language such as Ada, prevent proving that a program is correct. John Backus has proposed a functional language, FP, that is referentially transparent and overcomes this limitation.¹ FP has an associated algebra of programs that makes it possible to reason about programs by using the laws of this algebra. John Backus, K. M. George and G. E. Hedrick, and J. H. Williams have provided examples of program correctness and equivalence proofs using this

algebra of programs.^{1,2,4} One goal of this system is to provide the benefit of program proofs to portions of Ada projects that are originally written in FP, and then automatically translated to Ada. This allows Ada programs to use subsystems that have been proven to be correct.

Ada also provides the task type that allows programmers to manage explicit parallelism in their programs. Explicit control of parallelism becomes extremely difficult when there are large numbers of parallel processes. In contrast to the explicit parallelism allowed in Ada programs, functional languages, such as FP, contain large amounts of implicit parallelism—parallelism that is implied by the structure of expressions. The programmer does not control this parallelism, and thus, is not concerned with its complexity. The second goal of this system is to provide the benefits of parallel program execution to portions of Ada projects that are written originally in FP, then translated to Ada automatically. This is done by using Ada tasks to exploit the implicit parallelism inherent in FP programs.

An Overview of FP

This section presents an overview of the FP system that has been implemented at Oklahoma State University. It is similar to the FP system defined by Backus in his Turing Award Lecture.¹ Most of the differences are syntactic and were motivated by a desire to use this system with ASCII terminals. The FP system is composed of:

- a set of objects,
- a set of primitive functions that map objects to objects,

- a set of functional forms that produce new functions from existing functions,
- a definition facility for naming functions, and
- an application operator (denoted by $:$) that applies functions to objects.

Objects

An object is one of: an atom, a sequence of objects, or undefined (denoted by BOTTOM). An atom is one of: an integer, a boolean value (denoted by T or F), or the empty sequence (denoted by EMPTY). A sequence of objects is denoted by $\langle x_1, \dots, x_n \rangle$ where each x_i is an object. The empty sequence is also a sequence.

Primitive Functions

The following primitive functions are defined in this FP system. The functions are given on the left with their meanings on the right. (The symbol \perp is read *bottom* and ϕ is read *empty*).

$i: x \equiv x = \langle x_1, \dots, x_n \rangle \ \& \ 1 \leq i \leq n \rightarrow x_i; \perp$

$\text{length}: x \equiv x = \langle x_1, \dots, x_n \rangle \rightarrow n; x = \phi \rightarrow 0; \perp$

$\text{null}: x \equiv x = \phi \rightarrow T; x \neq \phi \rightarrow F; \perp$

$\text{id}: x \equiv x$

$\text{hd}: x \equiv x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 1 \rightarrow x_1; \perp$

$\text{tl}: x \equiv x = \langle x_1 \rangle \rightarrow \phi;$
 $x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow \langle x_2, \dots, x_n \rangle; \perp$

$\text{trans}: x \equiv x = \langle \phi, \dots, \phi \rangle \rightarrow \phi;$
 $x = \langle x_1, \dots, x_n \rangle \rightarrow \langle y_1, \dots, y_n \rangle; \perp$
 where $x_i = \langle x_{i1}, \dots, x_{im} \rangle$ and $y_j = \langle y_{j1}, \dots, y_{nj} \rangle,$
 $1 \leq i \leq n, 1 \leq j \leq m.$

$\text{and}: x \equiv x = \langle T, T \rangle \rightarrow T;$
 $x = \langle T, F \rangle \vee x = \langle F, T \rangle \vee x = \langle F, F \rangle$
 $\rightarrow F; \perp$

$\text{or}: x \equiv x = \langle T, T \rangle \vee x = \langle T, F \rangle \vee x = \langle F, T \rangle \rightarrow T;$
 $x = \langle F, F \rangle \rightarrow F; \perp$

$\text{not}: x \equiv x = T \rightarrow F; x = F \rightarrow T; \perp$

$\text{lt}: x \equiv x = \langle y, z \rangle \ \& \ y < z \rightarrow T;$
 $x = \langle y, z \rangle \ \& \ y \geq z \rightarrow F; \perp$

$\text{le}: x \equiv x = \langle y, z \rangle \ \& \ y \leq z \rightarrow T;$
 $x = \langle y, z \rangle \ \& \ y > z \rightarrow F; \perp$

$\text{gt}: x \equiv x = \langle y, z \rangle \ \& \ y > z \rightarrow T;$
 $x = \langle y, z \rangle \ \& \ y \leq z \rightarrow F; \perp$

$\text{ge}: x \equiv x = \langle y, z \rangle \ \& \ y \geq z \rightarrow T;$
 $x = \langle y, z \rangle \ \& \ y < z \rightarrow F; \perp$

$\text{eq}: x \equiv x = \langle y, z \rangle \ \& \ y = z \rightarrow T;$
 $x = \langle y, z \rangle \ \& \ y \neq z \rightarrow F; \perp$

$\text{ne}: x \equiv x = \langle y, z \rangle \ \& \ y \neq z \rightarrow T;$
 $x = \langle y, z \rangle \ \& \ y = z \rightarrow F; \perp$

$\text{add}: x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow y+z; \perp$

$\text{sub}: x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow y-z; \perp$

$\text{mul}: x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow yxz; \perp$

$\text{div}: x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \ \& \ z \neq 0$
 $\rightarrow y/z; \perp$

Functional Forms

The functional forms are actually second-order functions that accept functions (or objects in the case of constant) as arguments and produce a new function as the result. The following functional forms are provided by this FP system.

Composition

$(f \text{ compose } g): x \equiv f: (g:x)$

Construction

$\{f_1, \dots, f_n\}: x \equiv \{f_i: x, \dots, f_n: x\}$

Condition

$(p \rightarrow f; g): x \equiv (p:x) = T \rightarrow f:x;$
 $(p:x) = F \rightarrow g:x; \perp$

Constant

$(\text{constant } O): x \equiv x = \perp \rightarrow \perp; O$
 where O is any object.

Insert

```
//:  $x \equiv x = \langle x_1 \rangle \rightarrow x_1$  ;  
   $x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2$   
   $\rightarrow f: \langle x_1, //: \langle x_2, \dots, x_n \rangle \rangle ; \perp$ 
```

Apply to all

```
alpha  $f: x \equiv x = \phi \rightarrow \phi$  ;  
   $x = \langle x_1, \dots, x_n \rangle$   
   $\rightarrow \langle f: x_1, //: f: x_n \rangle ; \perp$ 
```

Function Definition

In this FP system, function definition is of the form

Def name = expr,

where name is an alpha-numeric string and expr is a function expression possibly containing primitive functions, defined functions, and functional forms. For example, one definition of the factorial function is:

```
Def SUB1 = sub compose (Id, constant 1)  
Def E00 = eq compose (Id, constant 0)  
Def FACT = E00  $\rightarrow$  constant 1 ;  
  mul compose (Id, FACT compose SUB1)
```

After these definitions have been provided, the application of the factorial function FACT:5 will yield 120.

An Overview of Ada Tasks

This section describes the Ada tasking facilities used by the FP run-time system. Further details on Ada tasks can be found in several books.^{4,5}

In Ada, the task type is provided to give programmers explicit control over the concurrent execution of programs. Each task is presumed to be executed on a logical processor of its own. These logical processors execute their tasks independently except at points where two tasks synchronize. In Ada, this synchronization is called a *rendezvous* and it occurs when one task calls an entry in a second task, and the second task accepts the call.

As one of the four forms of program units (the others are subprograms, package, and generic units), the task unit contains a task specification and a task body. The task

specification defines the entries (if any) for the task. The task body defines the executable portion of tasks of the type being defined.

As a limited type, a task type may be designated by the definition of an access type. This allows the dynamic creation of tasks, each with its own logical processor, by the evaluation of an allocator. It is important to note that the master of a task allocated in this way is not necessarily the unit that created the task. The master of a task allocated in this way is the unit that elaborated the definition of the access type. This has important implications as to the unit that is able to terminate a task and its slave tasks.

Synchronization and communication between tasks is accomplished by the *rendezvous*. As stated earlier, this occurs when one task calls an entry in a second task, and the second task reaches an accept statement for that entry. As an example of this consider the following.

```
task A is  
  entry BlockIn (P: in Block);  
end A;  
task body A is  
  -- declarations  
begin  
  -- do something  
  accept BlockIn (P: in Block) do  
    -- decode block  
  end BlockIn;  
  -- do something else  
end A;  
  
task B;  
task body B is  
  -- declarations  
begin  
  -- do some stuff  
  A.BlockIn (P);  
  -- do some more stuff  
end B;
```

Here we have task B calling an entry in task A. The *rendezvous* takes place when task B reaches the entry call and task A reaches the corresponding accept statement. The *rendezvous* continues until A has decoded the block and reaches the "end BlockIn" state-

ment. During the *rendezvous*, task B is suspended and does not resume until the *rendezvous* is completed.

Evaluation Strategy for FP Programs

Brief Introduction to Graph Reduction

Graph reduction is a popular method used in the evaluation of functional programs.³ To illustrate this technique, consider the following FP program and its evaluation via graph reduction.

Def $f = \{mul, add\}$
 $f: \langle 5, 7 \rangle$

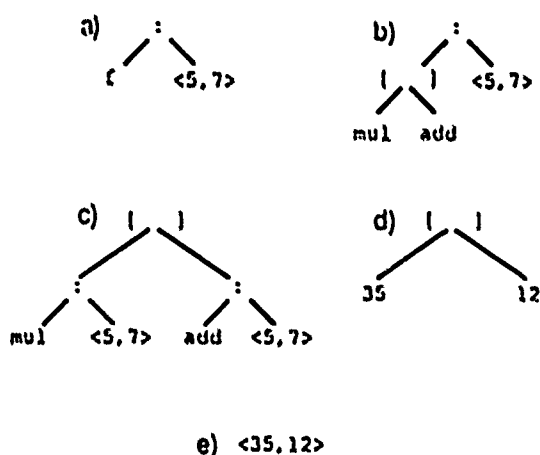


Figure 1

Figure 1 shows the steps that are performed by a graph reduction of the function f . In step *a*, the function is applied to its argument. Step *b* shows the expansion of f . In step *c*, the argument has propagated down to the leaf functions. Step *d* shows the reduction of two branches in the graph and step *e* shows the final reduction to the function result.

This simple example illustrates the graph structure of functional programs and their evaluation via graph reduction. It is important to notice that the reduction of independent sub-graphs can occur in any order without changing the result of the computation. The reductions performed in moving from step *c* to step *e*

could be applied from left to right, from right to left, or simultaneously. This property is the result of the referential transparency of FP programs and leads to their parallel execution.

The FP Run-time Evaluator

The FP run-time system exploits the parallelism revealed by the graph structure of FP programs. As an example of this, consider the graph of the FACT function defined earlier (see figure 2).

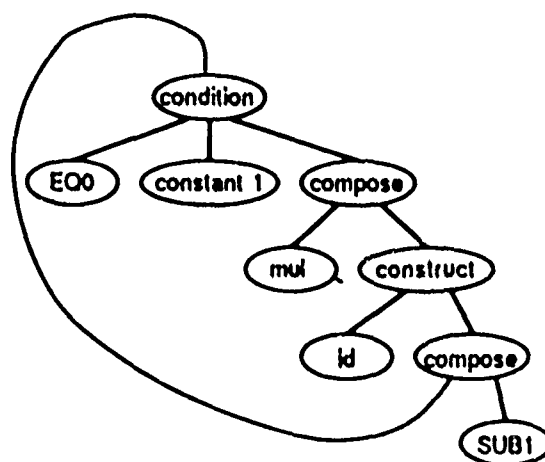


Figure 2

This graph guides the run-time system in the creation of a tree of tasks that computes the desired function. To start the process, the run-time system dynamically allocates a task and asks it to apply the functional expression rooted in the condition node shown at the top of figure 2 to the object of the computation. This task, and each new task that corresponds to an interior node in figure 2, is responsible for three actions:

- dynamically allocating tasks to compute its subexpressions (this corresponds to the function expansion shown by figure 1b),
- passing arguments to these new tasks (this corresponds to the argument propagation step shown by figure 1c), and
- collecting the results from these new tasks when their subexpressions have been computed (this corresponds to the reduction step shown by figures 1d and 1e).

The tasks corresponding to leaf nodes in figure 2 directly compute their result and return it to the parent task in a reduction step. At each reduction step, the corresponding task becomes terminated and disappears from the task tree. As reduction continues in the task tree, interior nodes become leaf nodes and are reduced. This process continues until only the root node of the task tree survives. At this point, the run-time system retrieves the result of the computation from the root node and execution is complete.

While the graph in figure 2 is cyclic, its corresponding task tree is always acyclic. A partial task tree corresponding to a possible evaluation sequence for the FACT graph is shown in figure 3. Some of the branches in the task tree have already been reduced.

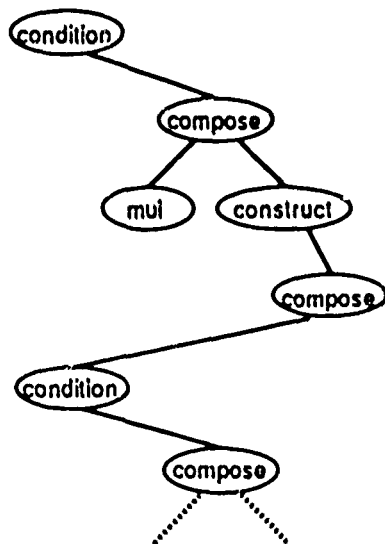


Figure 3

Of the functional forms in this FP system, only construction, condition, and apply to all have the potential to execute their child tasks in parallel. Unfortunately, the dynamic task allocation strategy that was chosen prevents the run-time system from taking advantage of the parallelism inherent in the condition functional form. The elaboration of the access type used in the allocation of new tasks occurs in the package specification of the run-time system. Thus the package that defines the run-time system is the master of every task created.⁴

This prevents us from killing one of the branches in the task tree rooted in a conditional node and raises the possibility of a runaway task branch. To execute the condition functional form in parallel, we need the allocating task to be the master of each task that it allocates.

Currently, the whole problem is avoided by simply ignoring the parallelism inherent in the condition functional form. A possible solution is to place a new access type definition for our task type in each task. Allocating tasks with the new access type should make the task dependencies match the conceptual task tree and allow the aborting of entire branches in the task tree.

Translation of FP to Ada

The FP to Ada translator is written in the C programming language and uses a Yacc generated parser and a Lex generated scanner. The translator generates Ada source code that builds objects and FP program graphs at run-time. Both the objects and program graphs are composed of nested lists. Each node in the list contains a tag that identifies the node's type and determines the node's contents. The translator places code at the end of the module that applies the main function to the main object and collects the result.

Project Status and Future Work

As it is implemented now, this system provides the possibility of program correctness and equivalence proofs for portions of Ada projects originally written in FP. To further this goal, this system needs a tighter integration of Ada to FP. In particular, Ada procedures should be able to construct FP objects and call FP functions directly with these objects as parameters.

The system automatically takes advantage of the implicit parallelism of FP programs by executing these programs in parallel, thus providing a potential for the increase of execution speed on a multi-processor machine. However, more work is required to exploit safely the par-

allelism of the condition functional form. Other possibilities for improvement include the development of code optimization via program transformation and the development of a heuristic algorithm that will prevent the creation of new tasks for computationally expensive functions. These improvements promise to improve the execution efficiency for FP programs.

References

- 1 J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *Communications of the ACM*, Volume 21, Number 8, August, 1978.
- 2 K. M. George and G. E. Hedrick, "Transformations in the algebra of functional programs," *Proceedings of IEEE Computer Society 1986 International Conference on Computer Languages*, October 27-30, 1986, Miami, Florida, pp 40-46.
- 3 S. L. P. Jones, "Using futurebus in a fifth-generation computer," Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK.
- 4 *Reference Manual for the Ada Programming Language*, "ANSI/MIL-STD-1815A-1983," New York, NY: Springer-Verlag, 1983.
- 5 R. Wiener and R. Sincovec, *Programming in Ada*, New York, NY: John Wiley & Sons, 1983.
- 6 J. H. Williams, "On the development of the algebra of functional programs," *ACM Transactions on Programming Languages and Systems*, Volume 4, Number 4, October, 1982.

Norman Graham received the B.S. degree in Computer Science in 1985 from Louisiana Tech University. He is currently pursuing a Ph.D. in Computing and Information Science at Oklahoma State University. His research interests include programming language design and implementation.

Mr. Graham is a member of the ACM and Upsilon Pi Epsilon, and is the recipient of the AMOCO Ph.D. Fellowship at Oklahoma State University.



TRANSFERRING FROM PASCAL OR C TO ADA

Jean Scholtz and Susan Wiedenbeck
Computer Science Department
University of Nebraska
Lincoln, NE 68583

ABSTRACT

This paper reports on a preliminary study of programmers trained in Pascal or C transferring to Ada. We videotaped subjects who spoke aloud as they wrote their first Ada program. We developed a model of programming in a new language and compared our observations to the model. As might be expected, we observed that a great deal of time is spent reading documentation and trying to translate the knowledge gained from the documentation into workable code. There was also a great deal of iteration of activities, as initial solution attempts failed and new approaches were tried. We found that subjects had little difficulty with the syntax of Ada or with the semantics of constructs which had close counterparts in their known languages. They did have difficulties with the semantics of unique constructs and with developing algorithms which would be easy to implement in Ada.

INTRODUCTION

The transfer of skill from one programming language to another is near transfer because it involves transfer between two skills which are closely related to each other¹. The opposite of near transfer is far transfer, which involves transfer from one skill or knowledge domain to another far afield from it. Studying near transfer between programming languages is of cognitive interest because it is an extension of the study of learning to program, an area in which much recent work has been done^{1,2,3}. In addition, there are practical reasons for interest in skills transfer in programming. Programmers at all levels from students to professionals are faced with the task of learning new programming languages, often on their own and without formal instruction. The current situation with Ada is a good example. Ada is seldom taught in university computer science programs, yet it is widely used in government and industry. Sometimes programmers are given in-house training in Ada, but not always, and the training courses vary widely in their length and content. Whether training is given or not, there is a strong underlying assumption that programmers can fairly easily transfer the knowledge and skills they have developed in other procedural languages, like Pascal and C, to Ada. Yet our experience teaching courses in which student programmers have transferred to a new language independently suggest that they encounter great difficulties. Although transferring to a new language is always easier than learning a first language, it is still a hard task.

Although there is no existing body of work on transfer of skills between programming languages, there is a growing body of research on near transfer in text editing. Much recent work on transfer of skills between text editors has been based on a common elements theory of transfer, i.e., the idea that knowledge of one editor will transfer to another only if the two share common elements^{4,5,6}. Taking off from the performance theory of Card, Moran, and Newell⁷, the common elements theory has been operationalized by defining the tasks in each editor by its own set of production rules. Quantitative predictions of transfer time can then be made by determining the number of new productions that

must be acquired to use the new editor.

In programming, unlike text editing, quantitative theories of performance and learning, which would be applicable to transfer, do not currently exist. In order to make theoretical progress we need a database of empirical evidence about what transfers and how. Our purpose in this study was to observe a small number of programmers transferring to Ada and use these observations to create a model of the activities programmers engage in and the strategies they use in transfer. We also wanted to characterize any special difficulties programmers seemed to face in beginning to use Ada. Because of the exploratory nature of the study and the small number of subjects, we did not intend to do any statistical analyses of the data.

EXPERIMENTAL METHOD

For this initial study we videotaped two programmers who normally work in Pascal or C as they wrote their first program in Ada. One of the subjects was a computer science graduate student and was a highly experienced and skilled student programmer. The other subject had worked as a professional programmer for three years in BASIC, COBOL, and Pascal. Both had previously learned new programming languages independently from books and manuals. The subjects were instructed to speak aloud freely as they worked, and their verbalizations were recorded. They were allowed to consult documentation, including an Ada textbook whenever they wished. The subjects were allowed a maximum of two hours. As expected, neither of them finished the program in that time.

Figure 1: RJE Problem

You are going to simulate a (simplified) version of a remote job entry facility which modifies and transmits data between two computers. Your program will do three things to each line of text read in:

1. Reverse the order of the characters.
2. Translate the upper case characters to lower case and vice versa.
3. If a line is less than 60 characters long, fill it with 'a's and write out a line of length 60. If a line is greater than 60, write it out as separate lines of exactly 60, filling the last line with 'a's if necessary.

The problem used in this experiment appears in Figure 1. It is a text processing problem which calls for a series of operations, algorithms for which would be familiar to intermediate and expert programmers. The solution to this problem in Ada has much the same outline that it would have in Pascal or C: a nested loop approach utilizing arrays of characters or strings. However, an ideal solution in Ada terms calls for using some features unique to Ada, in particular, packages for I/O and attributes for solving the case translation subproblem. Other Ada features which might be

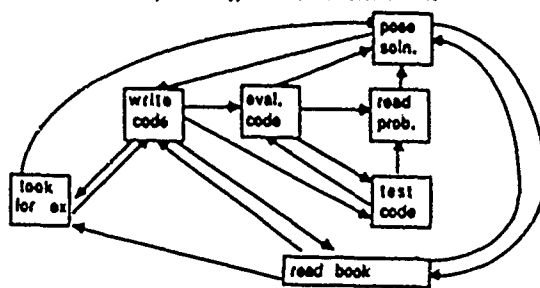
used but are not necessary for solving the problem are the initialization of variables when declared and the instantiation of generic types. We chose a problem which did not use the concurrency feature of Ada because we intended to use the same problem to study transfer to other programming languages which do not support concurrency.

MODEL OF TRANSFER

Our proposed procedural model for transfer between programming languages is shown in Figure 2. The model is quite general because the procedure we would expect the programmer to follow in transferring to a new language is not structurally different from the procedure he or she would follow when writing a program in a familiar language. The main distinctions between writing in a new language and writing in a familiar language would be the sequence of various activities and the number of times activities are iterated in carrying out a solution.

Our hypothesized model of transfer behavior is based on the idea that programs are written incrementally. One expects that a programmer writing a program in a language that he or she knows will first read the problem, then pose some sort of algorithmic solution. This will most likely be a language independent high level plan. Depending on how familiar the individual is with the programming language, he or she may go immediately to writing the code or instead may check the programming manual for specific information. One expects that as the programmer writes the code he or she may need to refer to the manual occasionally to check on such things as syntax, reserved words, and even semantics of constructs not frequently used. If the description in the book is unclear, the programmer may look for examples in the book or in previously written code. After the code is written, many individuals will check it by hand to insure that what is written is correct, but some will just attempt to compile and execute the code. After the code is tested, it must be evaluated to see if it is in error. If the code proves to be incorrect, the programmer will either pose a solution to fix it or read the documentation some more if no solution is immediately obvious. If no error is found, the whole foregoing process will be executed again to solve the next segment of the problem. The loop in Figure 2 will be reentered at either read problem or pose solution.

Figure 2: Hypothesized Procedural Model



If the programmer is using an unfamiliar language, we expect that the number of iterations of the write code-read book-look for example loop would be much higher. The read book-pose solution-write code loop would also be likely to occur with a higher frequency as the programmer attempts to implement a solution in the new language. If the new language is quite different from those that the programmer is used to, the frequency of these loops is likely to be increased more than if the new language is similar to those the programmer already knows.

OBSERVED BEHAVIOR

Figure 3 below shows the percentage of time each of our Ada subjects spent in each activity. Subject 1 was the graduate student programmer, and Subject 2 was the professional programmer. In the last column we have included for comparison the times for a subject who transferred from Pascal to Icon. Icon, a descendant of SNOBOL, is much less similar to Pascal than is Ada, so this Icon subject will allow us to make some observations on the effect of the language being transferred to on the transfer procedure.

Figure 3: Distribution of Solution Time (percentages)

	Ada S1	Ada S2	Icon
Read problem	1.8	2.0	3.5
Pose solution	5.5	13.0	21.3
Look for example	4.5	2.0	12.0
Write code	25.9	33.0	15.4
Test code	17.6	0.0	4.8
Evaluate code	13.0	2.0	4.5
Read book	31.6	48.0	34.0

The two Ada subjects spent a comparable amount of time in many of the activities, such as read problem, pose solution, look for example, and write code. However, they differed greatly in the amount of time that they spent on test code, evaluate code, and read book. It appeared to us that this difference was the result of different learning styles. Subject 1 was an active learner. He immediately began to write, test, and debug small segments of code. He only referred to the documentation when he reached an impasse and had no idea of how to proceed. As soon as he got some new idea from the documentation, he began implementing and testing it at once. Subject 2 was a more passive learner. He wanted to feel familiar with the language before trying to run even a small program, and he spent a good deal of time scanning the textbook to gain an overview. He spent about as much time actually coding as Subject 1, but he did not test and evaluate his code as he went. Instead he went on to solve successive subproblems, saving the testing for later (which he never reached because time ran out).

By comparison, the Icon subject spent less time writing code, more time posing solutions and more time looking for examples. Since he was less familiar with the new kinds of constructs used in Icon, he had to pose many solution plans before he succeeded in finding one he could implement in Icon. He also needed to consult more examples to see how the constructs should be coordinated into a program.

COMPARISON OF THE PROCEDURAL MODELS

Figures 4 and 5 represent the behavior of subject 1 and 2 respectively, in graphical form. When we compare these two figures with the hypothesized behavioral model in Figure 2 the following differences are indicated. Subject 2's behavior (passive learner) more closely resembles the hypothesized model except that Subject 2 did not engage in the test-code activity, and hence the hypothesized loops involving this activity do not appear in the actual behavior. A loop between read problem-read book appears in the behavior of both subjects which does not appear in our hypothesized model. The occurrence of this loop might be explained by the subjects' posing solutions in a language dependent fashion rather than attempting to use a high level plan and then attempting to implement it utilizing language constructs. The actual problem given to the subjects was such that the problem specification suggested a high level plan, so subjects did not explicitly state this but instead attempted to generate a more localized algorithm for dealing with individual portions of the problem. The pose solution activity for Subject 2 was never followed by looking for an example although it was followed by reading the book, which would suggest that his solution was specific enough that he could look up and read about constructs and syntax needed to implement it.

Figure 4: Procedural Model of Subject 1

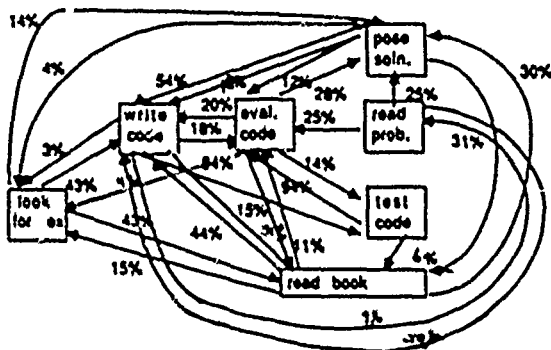
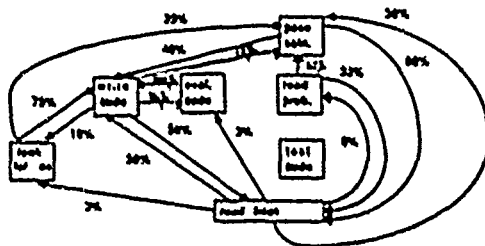


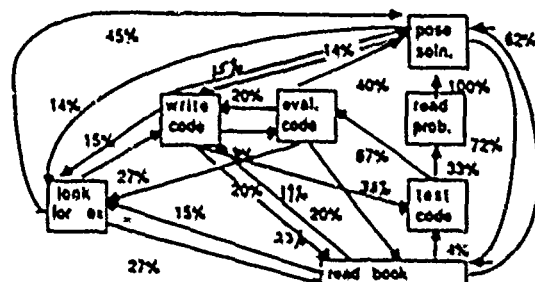
Figure 5: Procedural Model of Subject 2



Subject 1's behavior showed much more interaction between the various activities. Much of this can be explained by his active learner approach. His reluctance to read and look up many specifics, relying instead on "letting the computer figure it out", generated more iterations in loops and more paths than the hypothesized model did not predict. The paths from write code to test code, from test code to read book, and from evaluate code to look for example that did not exist in the hypothesized behavior model can be easily explained by the active learner behavior. The paths from read book to evaluate code, from evaluate code to read book, from test code to read book, and from evaluate code to write code that occur in one or both subjects' behavior and not in the model can be explained by the subjects' unfamiliarity with the language. Once code has been written they need to consult the book to see if the code appears to be correct and to verify the semantics of the encoded constructs. The experience level of the subjects did not seem to affect their behavior, although one would assume this might not be the case for novice subjects. This point will be investigated in later studies.

When we look at the behavior model constructed from the subject writing the same problem in the dissimilar language, Icon, one major difference is apparent. Figure 6 contains no path from read book to write code. Since the language being used in this case uses data structures and constructs quite different from those with which the subject was familiar, he demonstrated the need to explicitly state his solution. He did not feel that his usual implementation plan would work in this instance and tried to develop a solution more in keeping with the constructs available in the language.

Figure 6: Procedural Model of Icon Subject



PROBLEMS IN TRANSFERRING TO ADA

Ada is similar to other procedural programming languages in most ways, so major difficulties were not expected. This was especially true since our subjects were dealing with a simple programming problem for which they knew appropriate algorithms. To some extent this proved to be true. Particularly with syntax, few problems surfaced, and those which did were quickly resolved. For instance, subjects knew from brief scanning of the Ada textbook that looping structures such as while and for loops existed. When they needed to actually use these structures they efficiently looked up detail questions, for example, what is the begin-end equivalent in Ada. Another frequent approach to resolving syntactic questions was simply to code the construct as it would be done in Pascal or C and let the compiler locate mistakes. One subject took this approach when he wanted to find out whether array subscripts of 0 were allowed in Ada. He felt that it would be faster than looking it up in the documentation, and it probably was. The few syntactic questions which proved to be a bit more difficult to resolve came about because they were not explicitly answered in the documentation. For example, Subject 1 got an error message when using end_of_file. He hypothesized that he might need to type "end_of_file" in all uppercase or a combination of uppercase and lowercase. The question was not answered in the text, and the examples used were ambiguous. He had to experiment with all of the possibilities before concluding that his error had nothing to do with case.

Semantic problems were generally fairly minor, again because Ada is similar to other procedural languages which the subjects knew already. However, they did take on importance when programming constructs in Ada had no close counterpart in other known languages. This became very clear in our observations of the subjects using Ada packages. From overall scanning of the Ada textbook, the subjects had an idea of what packages were, and they quickly discovered that they had to use packages for I/O. However, they had trouble actually using them. One problem was discovering which package to use for I/O and figuring out whether you could use two packages at once. The other problem was not understanding exactly what operations were available in the packages and how they worked. For example, Subject 2 failed to discover the Get operation and had to restructure his original character-oriented solution approach to use the Getline instead. Subject 1 discovered the Getline but had great difficulty using it to read in strings because he was used to reading in primitive types in Pascal and did not think through what might happen in reading in structures.

The other area where we observed transfer problems was in planning solutions. The problem given to the subjects was chosen so that subjects would know algorithms for solving it. However, even though they knew algorithms they nevertheless had planning difficulties. The trouble was that the algorithms they initially proposed, though in a sense language independent, were based on what was convenient in other languages. For example, in solving

the problem of translating uppercase to lowercase and vice versa, Subject 1 took the approach (which would be appropriate in Pascal) of adding or subtracting the integer values of corresponding uppercase and lowercase characters. The method which he used for obtaining the integer value gave him an illegal type conversion error in Ada and forced him to abandon his algorithm. Subject 2 also tried to take the same Pascal-like approach of finding a function that would give a numerical value for a character. Although he scanned the textlook extensively looking for something that would help him, his search of the text was directed by what he knew of Pascal. As a result, he persistently looked for functions and failed to look at attributes, which would have solved his problem. We discovered many times while observing our subjects that, while in general their expertise with another language helped them, it also sometimes hurt them when it limited the solutions they would consider.

CONCLUSION

From this small sample it is impossible to draw statistically valid conclusions. However, we can note some interesting possibilities to watch for and some additional hypotheses to test as we do our more intensive studies. We observed difficulties in the ability of programmers to understand and know when to use constructs which had no direct analogies in their usual language. We also observed that their plans were based on algorithms that they knew would be easy to implement in their usual language. When these algorithms turned out to be awkward to implement in Ada, the programmers often had difficulty devising another algorithm. This suggests that if programmers understood new constructs better and had greater ability to pose alternative algorithms, then the iteration of paths within their behavior graphs would decrease. If this does indeed turn out to be the case, it would suggest that retraining of programmers focus on algorithm design and the semantics of unique constructs in Ada, leaving syntax and the remaining semantics for the programmers to deal with on their own.

REFERENCES

1. Bonar, J. and Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, 1, 133-161.
2. Card, S.K., Moran, T.P., and Newell, A. (1980). The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23, 396-410.
3. Olsen, E.W. and Whitehill, S.D. (1983). *Ada for Programmers*. Reston, VA: Reston Publishing Company.
4. Polson, P.G., Bovair, S. and Kieras, D. (1987). Transfer between text editors. *CHI '87 Proceedings*, 27-32.
5. Polson, P.G., Muncher, E., and Engelbeck, G. (1986). A test of a common elements theory of transfer. *CHI '86 Proceedings*, 78-83.
6. Salomon, G. and Perkins, D.N. (1987). Transfer of cognitive skills from programming: When and how? *Journal of Educational Computing Research*, 3, 149-169.
7. Singley, M.K. and Anderson, J.R. (1968). A keystroke analysis of learning and transfer in text editing. *Human-Computer Interaction*, 3, 223-274.
8. Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. (1984). What do novices know about programming? In *Directions in Human-Computer Interaction* (A. Badre and B. Shneiderman, Eds.), Norwood, NJ: Ablex, 27-54.
9. Spohrer, J.C. and Soloway, E. (1986). Analyzing the high frequency bugs in novice programs. In *Empirical Studies of Programmers* (E. Soloway and S. Iyengar, Eds.), Norwood,

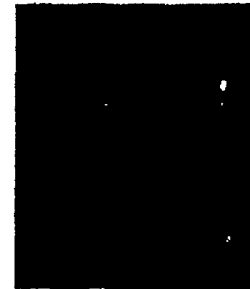
NJ: Ablex, 230-251.

AUTHORS



Jeanne Scholz
Computer Science Department
115 Ferguson Hall
University of Nebraska
Lincoln, Ne. 68582-0115
e-mail: JeanOSergvax.unl.edu

Jean Scholz is a doctoral candidate in Computer Science at the University of Nebraska-Lincoln. Her research interests include human-computer interaction, artificial intelligence, and programming languages.



Susan Wiedenbeck
Computer Science Department
115 Ferguson Hall
University of Nebraska
Lincoln, Ne. 68582-0115
e-mail: SusanOSergvax.unl.edu

Susan Wiedenbeck received her Ph.D. from the University of Pittsburgh in 1984 and is now Assistant Professor of Computer Science at the University of Nebraska-Lincoln. Her research centers on problems concerning the human-computer interface, particularly with respect to programming languages and text processing systems.

TWO APPROACHES TO ADA: THE PROCEDURAL APPROACH AND THE OBJECT-ORIENTED APPROACH

Gerald R. Thompson

Morehouse Software Group

ABSTRACT:

Pascal is often used as a reference point when approaching the Ada language. When a program developer used this procedural approach to Ada, he encountered problems of overloading and ambiguity. After learning an object oriented approach to programming, the developer rehashed some of his design and solved his problems. His opinion is that the procedural (PASCAL) approach, alone, is insufficient when approaching the Ada language. A knowledge of object-oriented programming is also required.

INTRODUCTION

This paper is based on one program developer's experience with Ada. It is not intended as some profound conclusion to years of research. It is a log of what happened when one program developer chose Ada as a development language. This is where the significance of the paper lies, for it is the program developer who must use the language.

DEVELOPER'S BACKGROUND

Since the Fall of 1983, the developer has been a member of a software development team. Over the years he has developed software for a variety of tasks. These include the development of: CAI packages for teaching Ada; graphics primitives for use with a microcomputer based PASCAL; a specialized data presentation tool for use with a mini-computer based statistical analysis package; and tools for automation of semi-repetitive tasks and documentation. The developer also has experience in developing software

for non-computer science purposes. He wrote and help design a computer model of the contrast sensitivity of retinal ganglion cells. The developer has experience with PASCAL, FORTRAN, ASSEMBLY, C, C++, and ADA. The developer had absolutely no experience in object oriented programming.

PROCEDURAL (PASCAL) APPROACH

The developer was faced with the task of writing a prototype of an authoring tool. He decided to test the Ada language on such a task. With no formal introduction to the Ada language or object-oriented programming, the developer used a procedural (PASCAL) approach. The PASCAL language is often used as a reference point when approaching Ada. This was the approach taken. The two languages are similar in structure and syntax; similar, but not the same. This was the source of initial problems. Struggling to learn the language and develop in it simultaneously was frustrating at first, but once the developer internalized the style of Ada, structure and syntax became intuitive.

The major problem occurred with the use of packages. When the developer tried to use more than one instantiation of the same generic package, he encountered overloading and ambiguity problems. In his particular case, he used more than one file of the same type. Intuitively he expected that when a read or write operation was requested and the file name was given, that the appropriate file information would be accessed. This was not always the case. In some instances the package names had to be provided. At the time, the developer thought that this was peculiar and that the file variable should be sufficient to determine the package.

THE OBJECT-ORIENTED (C++) APPROACH

During the summer, the developer was exposed to the C++ language. C++ is an object-oriented language. The

lessons learned studying this approach to programming would prove valuable. After extensive use of objects with this project, the developer looked at the design of the authoring tool in retrospect. In redesigning parts of the authoring tool, the developer avoided the problems of overloading and ambiguity by treating each file as an object.

CONCLUSIONS

The developer is even more impressed with the Ada language after learning an object oriented approach. In his opinion, Ada is an excellent software development language. Ada is oriented to both procedural and object-oriented aspects of the real world. Also, Ada's support for data with unknown constraints is ideally suited for real-world representation. Neither PASCAL nor the procedural approach alone are a sufficient prelude to the Ada language. It is important for a developer to understand object-oriented programming.



Gerald R. Thompson
Morehouse Software Group
P.O.Box 131
Morehouse College
Atlanta, Georgia 30314

Gerald is a Senior Computer Science major at Morehouse College. He has been a member of Morehouse Software Group for almost six (6) years. He has worked on special projects for E. I. Du Pont and Bell Labs. His experience also includes Biochemical research at Morehouse School of Medicine and Computer Modeling of some aspects of vision at Cornell University.

ON INCLUSION OF THE PRIVATE PART IN Ada PACKAGE SPECIFICATIONS

Sitaraman Muralidharan

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract

In Ada, if a package provides a *private type* (encapsulated type), it must also include a full declaration of the representation of the type in the *private* part of the package interface syntax (*package specifications*). Potential clients of a package are allowed to use all the information in the package interface syntax except the private part. This paper discusses the possible reasons for including the private part in package interfaces. The claim that the private part facilitates the compiler of a client program to generate efficient executable code is analyzed in detail. It is shown that the private part is undesirable, and that the elimination of the private part need not result in any serious execution inefficiency.

Introduction

Software reusability leads to two distinct programmer communities — the *developers* of the reusable software and the *clients* who reuse it. (The developer of one software component may be the client of another component.) This dichotomy necessitates that the specifications and implementations of reusable software components be kept separate. The specifications serve as a contract between the developer of a component and the clients.

Ideally, the specifications of a component should precisely state *what* the component does, and *nothing more*. An implementation should show *how* the specifications are realized in a particular way.

The specifications should provide a good *abstraction* of the specified concepts by exposing all the important details. Proper abstraction gives the developer the flexibility to realize the specifications

in many possible ways, and exposes to a potential client exactly what should be known to use the component, without having to be concerned about the actual implementation details. Specifications should also facilitate *information hiding*⁵, i.e., the specifications should reveal no more than what is necessary. The principle of information hiding complements the notion of abstraction. If the specifications state more than what is actually required, the developer is limited in what can be done to implement the specifications and the clients are overwhelmed with information that they need not know.

The separation of the specifications from implementations provides *developmental independence*, essential for software reusability, and for programming in the large. It is important that modifications to an implementation of a component do not affect the clients of the component (and hence the client components need not be recompiled), as long as the specifications remain unchanged. Actually, many possible implementations with different performance characteristics should be possible for the same specifications, and the clients of a component should have the flexibility to choose an implementation that suits their performance requirements. Ideally, a client program should not have to be recompiled even if it changes the implementation it chooses for a particular component.

Ada is one of the first widely-used languages to support the notion of separate component interfaces and implementations. Typically, an Ada package providing an abstract data type declares the type as a (*limited*) *private type*, and provides a set of operations to manipulate variables of that type. Every package has two separable parts — a package interface syntax and a package body. The package interface for an encapsulated type provides the name of the type, and the interface syntax for the operations on that type. In addition, it includes a *private* part which shows the representation used for the private type provided by the package, and this is the topic of discussion in this paper. The package interface syntax does not formally state what the package does, and hence should not be called

specifications. However, we will use the terms specifications (Ada terminology) and interface syntax interchangeably in this paper when there is no chance for confusion. The *package body* shows an implementation for the operations listed in the package interface syntax.

This paper examines closely the reasoning behind the inclusion of the private part in the interface syntax of a package, and shows that such inclusion — a clear violation of the principle of information hiding — results in no appreciable advantages.

Why the Private Part?

In this section, we critically examine the possible reasons for including the private part.

The Client Programmer Needs It

The private part of a package interface has been termed private, because it is not intended to be visible to the users of the package.

The clients of an Ada package are not allowed to use the information in the private part of the package interface syntax. In fact, clever environments should hide the private part from the clients of a package. While referring to the private part, Feldman⁴ notes that "unfortunately Ada requires that the *data structure* implementing a data type appear somewhere in the *specification* part of a package, and not in the 'hidden' part of a package, as we would like in the ideal" and that "For reasons having to do with how compilers for the language will be implemented, Ada compels us to write the *private* part in the *package specification*. Clearly, it would be preferable to hide those details away in the *body*." Obviously, the intention of the private part is not for use by client programmers.

The Compiler Needs It to Compile Client Programs

If some part of the package interface is not for use by potential clients, why is it there at all?

Apparently, the private part in a package interface is for use by the compiler of a client program. To make developmental independence possible, an Ada compiler should be able to compile a program just by referring to the interfaces of the packages the program uses, as long as certain *pragmas* (compiler directives) which require the compiler to refer to package bodies are not used. (The pragma `INLINE`⁶ makes the compiler expand procedure calls in line, and this will require looking at package bodies.)

It is possible that the private part shows the representation of the private type as an *access* (pointer) to a data structure that has not been shown in the package interface, and instead, has been

elaborated in the package body. In this case, the compiler of a client program essentially has the same information that it would have in the absence of the private part. If the private part were not specified, then the compiler can automatically represent a variable of a private type as an access to an arbitrary data structure, and the actual data structure can be filled in at run-time³. So the compiler can generate code for client programs without using the private part.

The Compiler Needs It to Generate Efficient Code

If it is possible to compile client programs without using the private part, then the reason for the private part must arise from considerations of execution efficiency of client programs³. The next section explores this possibility in detail.

Is The Private Part Needed for Generating Efficient Code?

We will now analyze whether the claim that a compiler can use the private part to generate efficient executable code for a client program is really true by considering various apparent uses of the private part.

Before making this analysis, however, we note that even if this is a valid reason, there is a much better solution than including the private part in the package interface syntax. Ada can introduce a new pragma which lets a compiler refer to a package body for the actual representation of a private type. Since Ada already includes the pragma `INLINE` which lets a compiler refer to a package body, inclusion of this new pragma should not raise any serious objections. This new pragma can be used after the developmental phase is complete.

An Example

We will use the generic 'Bounded_Stack' package¹ shown in Figure 1 as an example in this discussion. This package provides an abstract data type 'Stack'. A representation for the limited private type 'Stack' has been shown in the private part. Figure 2 shows a client program that uses this package. 'Int_Stack' is an instantiation of the generic 'Bounded_Stack' package with the item type 'Integer'. The procedure 'client' has a local variable 's' of type 'Int_Stack.Stack'.

```
generic
  type Item is private;

package Bounded_Stack is

  type Stack(size : Positive)
    is limited private;

  procedure pop(s : in out Stack);
  ...
```

```

private
  type Store is array
    (Positive range <>) of Item;
  type Stack(size : Positive) is
    record
      top : Natural := 0;
      contents : Store(1..size);
    end record;
end Bounded_Stack;

```

Figure 1 - Interface Syntax for the 'Bounded_Stack' Package

```

declare
  package Int_Stack is new
    Bounded_Stack(Item => Integer);
  use Int_Stack;

  procedure client is
    s : Int_Stack.Stack(size => 100);
  begin
    ...
    Int_Stack.pop(s);
    ...
  end client;

```

Figure 2 - A Client Program for the 'Bounded_Stack' Package

Direct and Indirect Representations

A compiler for the client program can make use of the information in the private part of the 'Bounded_Stack' package as follows. It can represent the variable 's' *directly* as a record containing an array of 100 integers, and a natural number. In the absence of the private part, the compiler could represent 's' *indirectly* as a pointer to a data structure (since the actual representation is available only in the package body), this data structure to be filled in later. The inclusion of the private part in the package interface syntax is because that there might be a slight execution advantage to direct representation.

If the representations of private types are known at compile-time, it is possible to allocate storage for variables of these types statically. Otherwise, the storage will have to be allocated dynamically. However, storage has to be allocated only once during the lifetime of a variable, and this cost may be amortized over the many operations on that variable. Furthermore, clever dynamic memory allocation techniques can be used in reducing this overhead¹. Hence, static storage allocation alone cannot result in any appreciable execution advantage. There is a possibility for some advantage when the representations are accessed. Let us explore this possibility further.

Representations of Encapsulated Types Cannot Be Accessed Directly: Direct representations will save an extra memory reference when the representation of a variable can be accessed *without calling a procedure*, for example, if the procedure 'client' can refer to 's.top' without calling any operations on 'Stack'. If 's' is represented indirectly, it will need two memory accesses to refer to 's.top', whereas only one memory access is needed if 's' were represented directly. But, this is not possible since the abstract data type 'Stack' has been encapsulated as a private type in an Ada package, and hence the only way to access the representation of a variable of that type is by calling one of the operations provided by the package.

In Figure 2, the procedure 'client' makes a call to 'pop', one of the operations on the encapsulated type 'Stack'. If the compiler has directly represented the variable 's' (using the information in the private part), it can either pass a pointer to 's' or can pass the actual representation. Ada leaves it to the compiler to decide on what is passed⁶. We will discuss both cases.

Parameter Passing by Reference: In our example, it is much more expensive to pass the actual data structure than a pointer to it, and hence a clever compiler will only pass a pointer to 's', even if 's' is represented directly. If 's' has been represented indirectly, then there is no choice. The pointer is passed to the procedure 'pop'. Thus, whether 's' has been represented directly or indirectly, what is passed to a procedure that operates on 's' is a pointer to the data structure representing the value of 's'. Thus, there is no execution advantage to direct representation, if parameters are passed by reference.

Gannon and Zelkowitz³ conclude from experimental evidence that "If data objects have to be passed (by reference) to procedures implementing abstract operations in order to perform the operation, the number of memory references in the direct and indirect implementations is likely to be similar." They propose that the compiler expand the procedure calls on encapsulated data types in line, and thus eliminate the indirection that arises from procedure calls even when direct representations are used. However, if a compiler has to expand procedure calls in line (for example, when the *pragma* *INLINE* has been used), then it must refer to the package bodies of the corresponding packages. If a compiler must refer to the package bodies to improve the execution efficiency in any case, then there is no need to make the representations of private types available to the compiler through the private part in package interfaces.

Parameter Passing by Value-Result: If parameters are passed by value-result, then the representation for the parameter has to be copied, and this takes normally time proportional to the size

of the representation. Hence, reference parameter passing is usually more efficient for most data structures. However, if the representation of an encapsulated type is very small (e.g. an integer), then its representation might be profitably passed by value. (Ada insists that parameters of scalar and access types be always passed by value-result⁶.) This would be possible, only if the compiler knew the representation for the private type.

When the representation is indirect, only a pointer can be passed as a parameter. However, the called procedure can dereference this pointer at the beginning, and can use the dereferenced address thereafter, whenever this parameter is accessed. This is possible since the called procedure knows the actual representation. Dereferencing takes one memory cycle and, such dereferencing is necessary only in the case of *in* and *in out* parameter modes.

We conclude that whenever the representation of a private type is comparable in size to that of a pointer, and an operation is called on that type, there is an extra memory reference if the private part is not used. But, common sense and experience would argue that this overhead is very small in actual practice.

Overheads from Access Checks: One of the findings of Gannon and Zelkowitz³ is that when an indirect representation is used, the compiler needs to generate code to ensure that the pointer to the accessed data representation is not null. They report that this overhead from indirect representation is negligible when a pragma to suppress the access check is inserted.

One possible solution to this problem is to ensure that representations of variables of private types are *never* null. A good design practice is to require that every Ada package must provide two operations "initialize" and "finalize" for each private type provided by the package. The "initialize" operation for a private type would allocate storage for the representation of a variable of the type, and initialize the representation appropriately. The "finalize" operation would reclaim the storage. In our example, in the client program, immediately after the procedure 'client' is called, the representation of the local variable 's' should be initialized with a call to the operation "initialize." Similarly, just before the procedure finishes execution, the storage for the representation of 's' could be reclaimed with a call to the operation "finalize." These operations could be invoked automatically by compiler-generated code whenever a variable of a private type provided by the package is used. The language should establish these rules, and eliminate a common source of error in programs resulting from misuse of pointers.

Why Eliminate the Private Part?

Even if it is convincing that there may not be any significant advantages from the inclusion of the private part in the package interface syntax, why is it any harm? This section discusses the many advantages that result from the elimination of the private part.

Improved Abstraction

While considerations for a certain implementation may usually guide the design of specifications to some extent, it is unnecessary to let implementation considerations take priority over the specifications of an abstract concept. Having to specify the private part in the package interface may overly bias the interface towards a particular implementation. Such an approach may result in interfaces which are far too removed from the specified concept, and too close to an actual implementation.

It is also conceivable that the interface syntax of a package and the actual implementations may be developed by different people. In such a case, it is certainly undesirable to let the specifier do part of the implementation². Implementation decisions taken too early may turn out to be wrong.

Better Information Hiding

The principle of information hiding is that the specifications should expose no more than what is necessary. As pointed out earlier, over-specification restricts the choices of a developer in implementing the specifications, and overwhelms a client with superficial information that is not actually needed to use the component. In Ada, the private part in a package interface exposes more than what is needed. One common suggestion that is advanced to protect the clients from this superficial information in Ada package interfaces is to build clever environments to hide the private part from the clients. However, this suggestion can be extended, and it can be argued that even the actual implementations of the operations could be listed with the operation interfaces, and it should be left to the environments to hide the implementations from the clients! Such an approach combines the interfaces and implementations, and is obviously undesirable.

Improved Developmental Independence

The specifications of a package are like an agreement between the developer of the package and the clients. As long as the specifications are unchanged, the developer is free to change any actual implementation detail. For example, in Ada packages, a developer can change the implementation of one or more operations provided by a package by changing the package body, without

affecting clients of the package. That is, changes in implementations of the operations provided by a package do not result in recompilation of the client programs that use the package. However, a change in the representation of the type provided by a package makes the developer change the private part in the package interface, and thus forces recompilation of the client programs which use the package. Recompilation is certainly undesirable in large systems, and removing the private part will totally eliminate redundant recompilation.

Multiple Implementations for the Same Specifications

If the specifications do not expose representational details in the private part, then it is possible to implement the same abstract data type in many different ways. For example, the abstract data type "bounded stack" could be represented using an array or a linked list. Since in Ada, it is possible to associate only one package body with a package specification, it will be required to define two packages with the same specifications to support two different implementations. However, it is a valuable feature if a language supported multiple realizations for the same specifications. It is possible that the inclusion of the private part in the package interfaces might have been an important reason for not allowing multiple implementations in Ada.

Conclusions

The representation of a private type provided by a package does not belong in Ada package specifications, and the private part which shows this representational detail is unnecessary. The private part violates the principles of abstraction and information hiding, hinders developmental independence, and does not allow the possibility of multiple implementations for the same specifications. Elimination of the private part, while solving these problems, need not result in any serious execution inefficiency. The representation of a private type provided by a package belongs in the package body, and that is where it should be.

Acknowledgments

I am pleased to acknowledge the contributions of Prof. Bruce Weide towards this work. He illuminated some of the key issues, offered many valuable suggestions, and carefully reviewed this paper. I would also like to thank Suresh Sitaraman and Prof. Prasad Vishnubhotla for their helpful suggestions.

References

1. Booch, G., *Software Components with Ada*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1987.
2. Brooks, Jr., F. P., *The Mythical Man-Month*, Addison-Wesley Publishing Company, Menlo Park, California, January 1982.
3. Gannon, J. D. and Zelkowitz, M. V., "Two Implementation Models of Abstract Data Types," *Computer Languages*, vol. 12, no. 1, January 1987, pp. 21-25.
4. Feldman, M. P., *Data Structures with Ada*, Reston Publishing Company, Inc., Reston, Virginia.
5. Parnas, D. L., "A Technique for Software Module Specification with Examples," *Communications of the ACM*, vol. 15, no. 5, May 1972, pp. 330-336.
6. *Reference Manual for the Ada Programming Language*, United States Department of Defense, Washington, D.C., January 1983.



Sitaraman Muralidharan is a Ph. D. student and a research assistant in the Department of Computer and Information Science at The Ohio State University. His research interests are in distributed systems, programming languages, and software engineering.

Muralidharan got his B.E.(Honors) Degree in Electrical and Electronics Engineering from the Regional Engineering College, Tiruchi, University of Madras, India in 1983. He got his M.E.(Distinction) Degree in Computer Science and Automation from the Indian Institute of Science, Bangalore, India in 1984. His Internet address is murali@cis.ohio-state.edu, and his postal address is Department of Computer and Information Science, 2036 Neil Avenue Mall #228, Columbus, Ohio 43210.

What is the Object in Object Oriented Programming

Keng Voon Chan and Wang Tsung-Juang

University of Mississippi
Student Paper

ABSTRACT

This paper examines the two languages, Ada and Smalltalk, and provides a comparison of the nature of the object in each. Smalltalk's only data structure is the object and programming is a matter of message sending between them. Ada, on the other hand, is an ALGOL-like block structured language which supports all of the simple scalar types and all of the structured types available in most block-structured languages. The object in Ada is in terms of data abstraction available through the Ada package and separate compilation. This paper illustrates, through the use of simple little procedures, the differences in the two approaches. Finally, through a careful evaluation of Smalltalk it tries to show what is meant by an object-oriented language.

The object-oriented style has often been advocated for simulation programs, system programming, graphics and AI programming. Object-oriented programming, or sometimes called message-passing programming views data as objects. An object is an entity which combines the properties of procedures and data since both perform computations and save the local state. Objects respond to messages using their own procedures called "methods" for performing operations. An important aspect of programming used by message sending is Data Abstraction which means calling programs should not make assumptions about the implementation and internal representation of data types that are used by them. Their purpose here is to make it possible to change underlying implementations without changing the calling program.

By a protocol we mean messages used to define a uniform interface to objects which provide a particular facility. Additional leverage is provided for building a system when the protocol is standardized which is possible because of polymorphism. Polymorphism in a computer system refers to the capability of different classes of

objects to respond to exactly the same protocols.

Inheritance is another important concept in object-oriented programming languages. The properties which allow for inheritance include methods, class and instance variables. Multiple inheritance is the ability to inherit properties from two or more classes and the accompanying semantics are an area of current research.

A typical object-oriented, message-passing language will view data as contained in objects. Each object contains representation information, and defines the types of manipulation that may be performed upon the object. This implies that objects are strongly typed. A message which specifies the operation and the operand is sent to the object. The object then determines whether or not it knows the message type. If it does the operation is performed and the object if required is returned.

The object model is an abstraction mechanism useful for understanding the design and implementation of systems. An object is either active or passive. Passive objects are program variables or databases. An active object is instantiated in a program or procedure which in turn transforms or acts upon passive objects. Basically an object is an entity that combines the properties of data and procedure and save the local states.

A procedure-oriented language consists of a main program and possible calls to subprograms which are procedures and functions. Data is shared by subprograms through several techniques, including passing parameters in the subprogram invocation. The basic view is that there is data and there are procedures that manipulate the data. While Ada can support object oriented design because of its emphasis on data abstraction its approach to an object is fundamentally different.

In an object-oriented language, each object is an instance type which refers to a class or an instance variable or a method which manipulates an object or a temporary variable which issued by the instantiation of a method. Some object-oriented languages permit the declaration

of class variables which are shared by objects of the class in which they are defined.

The first substantial interactive, display-based implementation of an object-oriented programming language was Smalltalk. The most popular version of this is Smalltalk-80 licensed by Xerox.

In Smalltalk most objects are divided into classes and instance variables. A class is a description of one or more similar objects. "Instance" is a term used to describe either the relation between an object and its classes or as a noun referring to objects that are not classes. Smalltalk as a language was produced as part of a Dynabook project initiated by Allen Kay in the Learning Research Group at Xerox Palo Alto Research Center.

Smalltalk is implemented by passing a message to an object. Each expression that is evaluated results in a message being sent to a receiver. The receiver which evaluates the message selector, determines the method. There are three types of message selectors: unary, binary and keyword. In Smalltalk, an object's class determines how a selector is to be interpreted. The binary messages +, -, * and / may take on different meanings depending upon the class of the receiver. For instance, the selector + can be used to add integers or perform union of sets. The interpretation is entirely dependent upon the method of an object. This might be compared to overloading an operator in Ada. The critical difference here is a binding time issue. The Smalltalk programming environment tries to provide every tool for finding, viewing, writing and running the methods. Everything is inside a window. Every program is just a part of the whole system that is linked together.

In Ada, in order for a procedure to call a variable, the name declaration binding has to be specified e.g.:

```
x.getname(A);
```

where A is declared as a character earlier in the program or

```
x.getname(B);
```

where B is declared as an integer in the declaration section of the code.

This kind of binding is called static binding. This means that the type of the argument, that is, the type of argument A or B above is determined when procedure x.getname is compiled with argument A or B. In contrast, in object oriented languages, the name of an object is not bound to a particular class until runtime. The use of methods allows an operation to be available for the instance of a class when a message is sent to an object and its "type" is bound at the moment of the message passing and not before. For example:

```
x.getname(A or B)
```

means to get A or B the message getname,

sent to an object of class x. Sending a message is very similar to calling a procedure. The difference occurs when the message is received by the object. It is the object that decides which method is to be executed. This occurs when the message is sent which happens while the program is executing hence the dynamic binding.

This difference in binding is the most critical difference between Ada and object oriented programming languages. Of course the obvious differences of compiled code versus interpreted code is an issue but it is more obvious than the binding time issue.

The similarity of the two languages lies in the way that they deal with data abstraction or the ability of each to encapsulate an object and its particular attributes. Ada, through its package feature, allows the user to write and use fully encapsulated object descriptions giving to the user only the features of the object which the user is required to have in order to use the object. Smalltalk likewise is able to fully encapsulate an object and to allow the object to inherit from other classes many features. It can limit the user's view of the object through the use of inheritance. It is here that the comparison ends. Ada is a completely procedural language designed to run on Von Neuman machines with tasking added to allow for concurrency. Smalltalk has a completely different design philosophy. In the design of Smalltalk the basic principle was not how a machine worked but rather how people thought about solutions to problems. One of the principle design goals of the authors was to provide an easy interface between users and machines, this was done with objects being the primary basis for the interface. Humans naturally solve problems in terms of objects hence, object oriented languages consider the object and its relation to objects as the design issue. It is this issue which is the motivating force in software engineering which considers itself to be object-oriented. The design process is object oriented, the programming process, in non-object oriented languages, is not.

Software engineering, like object-oriented programming, requires an environment in order to develop large pieces of code. Ada does not provide that environment but it has been the catalyst for the development of fully integrated software development tools. These tools often function much like the environment provided by many object-oriented languages but the programming language process itself is very different using the two different languages.

To graphically illustrate the differences between the two languages let us look at two identical programs, one in Ada and one in Smalltalk-80. The first example, in Ada, is taken from Gehani's book Ada an Advanced Introduction p. 63:

```
with Test_10; use Test_10;
procedure TOWER_OF_HANOI is
package IO_Integer is new Integer_10(Integer);
use IO_Integer;

Number_of_Disks:Natural;

procedure Hanoi(N:Natural; X,Y,Z: Character) is
begin
  if n /= 0 then
    Hanoi(N-1,X,Y,Z);
    Put("Move disk ");Put(N);
    Put(" from ");Put(X);Put(" to ");Put(Y);
    New_Line;
    Hanoi(N-1,Z,Y,X);
  end if;
end Hanoi;
begin
  Put("How many disks have to be moved?");
  New_Line;
  Get(Number_of_Disks);
  Hanoi(Number_of_Disks, 'X','Y','Z');
end Towers_of_Hanoi;
```

A program which accomplishes the exact same thing is written below in Smalltalk-80, this code is taken from a book which contains a tutorial for Smalltalk-80 by Ted Kaehler and Dave Patterson A Taste of Smalltalk (p.7 or p. 23). It is important to realize here that this code would run in the "windowed" environment of the Smalltalk-80 system so that it is not so clearly understandable when presented purely as code.

Example:

```
moveTower:height from:fromPin to:toPin using:usingPin
"Recursive procedure to move the disk at a height
from one pin to another pin using a third pin"
height > 0 ifTrue:
  self moveTower:(height-1) from: fromPin to:usingPin using:toPin.
  self moveDisk:fromPin to:toPin.
  self moveTower:(height-1)
    from:usingPin to:toPin
    using:fromPin
moveDisk:fromPin to:toPin
  "Move disk from a pin to another pin. Print the results in the
  transcript window"
  Transcript cr.
  Transcript show:(fromPin printString, ' -> ', toPin printString).
```

In order to run the program above a call statement must be issued, it might be the following:

```
(Object new) moveTower:3 from:1 to:3 using: 2.
```

In the Smalltalk-80 program above the method is moveTower, the objects are fromPin and toPin, the program messages are self moveDisk and self moveTower where moveDisk and moveTower are messages sent to transcript which is a window in the system browser. Self is a Smalltalk-80 reserved word. The flow of the program is recursive in the same sense as the Ada program, it is just that in the Ada program the "objects" are X,Y,Z and N but note that they are actually characters and numbers where in

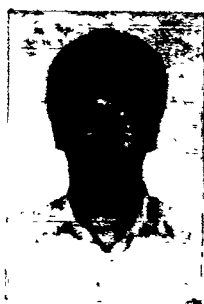
the Smalltalk-80 program the objects are simply objects and it is not really apparent to the user just what they are. The user is freed from the process of matching some non-disk data with the object of interest, in this case disks on towers. So that true data abstraction is available in the Smalltalk-80 program in a sense in which it is only approximated in Ada.

In conclusion, it is now clear that while there are some similarities in the two programming languages the differences are vast and critical. Smalltalk is a system that is composed of objects which can be executed independently and in parallel with all other existing objects, where the objects interact with one another by passing and receiving messages. Therefore the concurrent object-oriented approach has a lot in common with the process model in Ada.

Most experienced programmers will have problems learning Smalltalk-80 or other similar object-oriented languages because they are new and philosophically so different from traditional procedural programming languages. Perhaps object-oriented design and object-oriented programming share this difficulty in learning. They are perhaps too close to how we actually think for us to see how correct they are for designing good computer software.

REFERENCES

1. Narain Gehani, Ada An Advanced Introduction, Prentice-Hall, Inc., Copyright 1983.
2. Ted Kaehler and Dave Patterson, A Taste of Smalltalk, W.W. Norton and Company, Inc., Copyright 1986.



Keng Yoon Chan is currently a Senior in the Department of Computer Science at the University of Mississippi. He expects to receive a Bachelor of Engineering Science in May, 1989.



Wang Tsung-Juang is currently a Junior in the Department of Computer Science at the University of Mississippi. He expects to receive a Bachelor of Engineering Science in May, 1990.

A TWO-PHASE REPRODUCTION METHOD FOR Ada TASKING PROGRAMS

Mamdouh M. Najjar and Tzila Elrad

Computer Science Department
Illinois Institute of Technology
Chicago, Illinois 60616

ABSTRACT

Different results are produced when an Ada tasking program is re-executed with the same input due to two types of nondeterminism. This problem exists in cyclic debugging of Ada tasking programs. Nondeterminism reproduction is difficult in Ada due to some Ada characteristics. Our approach uses a preprocessor to extend an Ada tasking program P using a path specification S into P' such that P' is a deterministic version of P. P' can then be re-executed as many times as required by the debugger to locate the source of an error. Each phase handles a different type of nondeterminism. Phase One creates one Ada task controller per task. Each controller handles the arrival sequence of entry calls to its assigned task. Phase Two handles nondeterministic selections by controlling the selection of alternatives within selective wait statements. One advantage of this approach is that it uses more than one Ada task controller for the reproduction process. This eliminates the need for a master controller which can be a bottleneck to a solution. The two phases are easy to understand and to implement.

1. INTRODUCTION

There are basically two types of nondeterminism that cause Ada tasking programs to produce different results for the same input every time they are executed. Global nondeterminism arises as a result of the relative progress of tasks within a program, and local nondeterminism arises as a result of an explicit choices of a nondeterministic control structure [10, 15]. Reproducing the same results from the same input in a

language that supports one or both of these types is called the reproduction testing problem [4, 21]. Reproducing concurrent programs normally requires the reproduction of the two types of nondeterminism. Global nondeterminism is usually more difficult to reproduce than local nondeterminism. This is due to the fact that global nondeterminism is difficult to record. Recording an execution sequence for independent events in different tasks is an example of recording global nondeterminism. On the other hand, local nondeterminism is local to each task and can easily be recorded and replayed. The problem exists in cyclic debugging of concurrent programs.

Cyclic debugging is a well known process for debugging sequential programs. It is used to locate and remove errors after they have been uncovered by a test case. This process is well understood for sequential programs but not as well understood for concurrent programs. The same process has been adopted for concurrent programs [17]. Cyclic debugging of Ada tasking programs cannot be achieved without being able to reproduce the same results from the same input. Locating and removing an error usually requires more than one execution. This requires finding ways for reproducing the two types of nondeterminism mentioned earlier. Solutions for nondeterminism reproduction are different for different languages because of the characteristics of the interprocess communications and to the nondeterministic control constructs supported by a language.

Reproduction of global nondeterminism for an Ada program is reduced in this paper to the reproduction of rendezvous (no reproduction is done

for shared variables). The reproduction of local nondeterminism is reduced into the reproduction of nondeterministic selections within an Ada task due to a selective wait statement.

Reproduction of a rendezvous requires that the two partners of a rendezvous match. In languages that support the symmetric naming convention, i.e., the called task knows the names of its callers and vice versa, a construct for matching the two partners of a rendezvous is usually built into these languages or done automatically [14]. Reproduction of a rendezvous in such languages is obviously easier than in those languages that adopt the asymmetric naming convention (the called task does not know the names of its callers). We expect that rendezvous reproduction in Ada will be difficult for a number of reasons: Ada adopts the asymmetric naming convention, Ada handles entry queues in strictly first-in first-out order, and Ada does not have a mutual control construct, i.e., accepting an entry call according to some value of its passed parameters.

A solution to the reproduction problem for Ada transforms an Ada program P into P' such that the reproduction of the same results of P requires one execution of P' with an additional input of a rendezvous sequence which represents the previous execution of P [21]. The solution is based on the reproduction of a rendezvous sequence using a controller that controls the arrival of entry calls to the called task. Each entry call must first call a controller and identify its source and destination; then the controller returns the call when the source is the other partner of the next rendezvous in the destination task. The next entry call to the next rendezvous is released by the controller when the previous rendezvous has started. One drawback of this method is that a centralized controller, which can be a bottleneck to the program, is used.

Some approaches for debugging concurrent programs avoid the problem by building a debugger that has the ability to discover and locate errors

or to record the program's state at each stage of the execution [1, 9].

Another approach suggests using a new programming construct called preference control to control the race conditions within Ada tasks [8]. This method handles only local nondeterminism and does not force a selection; rather it suggests one. Other related non-Ada work is presented in several references used for this paper [1,17,18,20,22].

This approach basically reproduces a program's rendezvous sequence by reproducing all task's local rendezvous sequences. A local rendezvous sequence is associated with a task in an Ada program. This approach is partially based on the theoretical work given in the following references: [2,6,7,15,16]. The approach suggests reproducing each local rendezvous sequence independent of the other local rendezvous sequences to reproduce an original behavior of a program.

In this approach, a controller is used to simulate a communication environment for each task in the original execution. The selection of a recorded sequence of nondeterministic decisions a task has taken is enforced. As a result, a task behaves in the same way it did in the original execution.

The approach is divided into two major phases. Phase One, which handles global nondeterminism, uses an Ada Task Controller (ATC) per task to control an arrival sequence of entry calls to a called task. It insures that the order of entry calls at each entry's queue is in a predetermined order. The second phase, which handles local nondeterminism, controls nondeterministic selections within individual tasks. This is done by using a set of conditions to disable or enable rendezvous in a selective wait statement. Using these conditions one can enable the next rendezvous of a rendezvous sequence. These two phases distinguish between the two types of nondeterminism mentioned earlier and handle each type separately. Note that each of these two phases requires some extensions to the Ada source program, i.e., the addition of some special Ada code to

the original program. This extension process is referred to as extending a program.

One advantage of this approach is that it uses one or more ATCs. One ATC is assigned to control entry calls to one or more tasks. This simplifies the implementation of ATCs and eliminates the need for a master controller, which can be a bottleneck to a solution.

The design of an approach that can be divided into two smaller phases is another advantage of this approach. Each phase deals with a problem in the reproduction process, but the two phases work together to achieve the goal. When a problem is spotted at the reproduction process, the nature of the problem guides us to the phase in fault. Other advantages include better understanding of the reproduction process and ease of implementation.

This approach limits handling of local nondeterminism constructs by using the selective wait statement (the other types of select statement are not handled) and by assuming that no rendezvous nesting occurs in it. The COUNT attribute and shared variables are also not handled. An approach for handling these constructs is given in reference [21].

A discussion of problem of reproducing an Ada tasking program and what special Ada characteristics may influence the solution is provided in Section 2. Outlines of the reproduction process are given in Section 3. Explanations of the two phases of this approach are presented in Section 4. A complete example is given in Section 5, and conclusions are presented in Section 6. Appendices I and II list the Ada code for the example given in Section 5.

2. REPRODUCING ADA TASKING PROGRAMS

In languages that adopt a symmetric naming convention, rendezvous can be reproduced easily; the two partners of a rendezvous automatically match in Communicating Sequential Processes (CSP) [14]. In CSP, rendezvous match through an Input and Output commands: a caller issues an Output command specifying its destination task, and

the called task issues an Input command specifying its source task [14]. In Ada, rendezvous are more difficult to reproduce because an entry in a destination task (a called task) should rendezvous with the first call at its queue regardless of who is calling it, and each entry queue is handled in strictly first-in first-out order. This implies that a destination task cannot determine where each call originates. If the name of the calling task (source task) is included as an entry call parameter, the destination task does not find out the name of its caller until the rendezvous has begun. Because Ada does not have a mutual control construct, this restriction cannot be avoided.

A mutual control construct can be useful for solving the problem of mismatching the two partners of a rendezvous in Ada. Such a solution requires either simulating mutual control by the existing Ada constructs or adding such a construct to the language. However, a mutual control construct is not needed if we are able to duplicate every entry queue order in the reproduction execution. This can be done by duplicating (replying) the original arrival sequence of entry calls to a destination task, and as a result, every entry queue sequence in a the destination task is duplicated. This approach diminishes the need for a mutual control construct for the purpose of reproduction of rendezvous in Ada. Such a construct will still be useful for explicit scheduling of Ada tasks. This approach adopts the approach of duplicating the arrival sequence of entry calls to a destination task. This is done by using an ATC per destination task to control its arrival sequence of entry calls.

3. REPRODUCTION PROCESS OUTLINE

Figure 1 presents the general outline of the approach. A path specification is a set of local rendezvous sequences that were recorded in a previous execution, usually one local rendezvous sequence per task. A path specification file is a file that contains a local rendezvous sequence for each non-

calling task (a task with at least one accept statement that produced a rendezvous in the original execution). A preprocessor command file is used to specify the number of ATCs that will be used in a program and to specify which task is controlled by which controller. An Ada source program, a path specification file, and a preprocessor command file are used as inputs to a preprocessor. A preprocessor is a program that extends an Ada source program to accommodate new Ada code for reproduction. The new added code includes the number of ATCs specified in the preprocessor command file (ATCs are explained later in this section).

The output of a preprocessor is an extended Ada program that has the same semantics of the original Ada program. The differences are the following: the extended Ada program is deterministic, it always follows the same path (specified in the path specification file), and it gives the same results for the same input every time it is executed. When the extended Ada program is executed, the results of its execution will be compared to either the expected results or the results of the original execution. Modifications can then be made to the original Ada program, if desired, and the whole process can be repeated until the desired results are achieved.

Definition 1:

A local rendezvous sequence for a task T is a totally ordered sequence of rendezvous accepted by task T (T is the destination task), where each rendezvous is represented as a three-entity tuple:

< Rendezvous sequence number,
Calling task identity, Entry name >

A rendezvous sequence number is unique within a local rendezvous sequence of task T. Note that from this definition we can conclude that a task with one or more accept statements which produces no rendezvous in the original execution, or one with no accept statements at all has no local rendezvous sequence. We call such a task a demanding task.

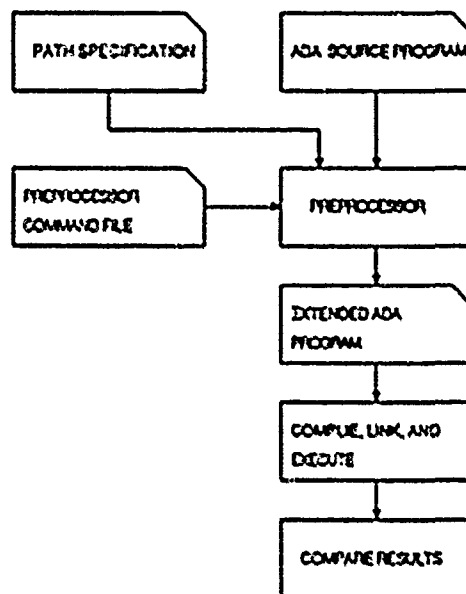


Figure 1. An outline of this approach.

4. TWO-PHASE REPRODUCTION FOR ADA TASKING PROGRAMS

Two phases of program reproduction—an entry calls control phase and a nondeterministic selections control phase—are discussed in this section. The extension procedure for each phase is also presented.

The first phase is to use one Ada task controller per task (or a group of tasks) to control the arrival sequence of entry calls for that task. An ATC assigns a unique number, called entry call sequence number, to entry calls that are calling the same entry. Entry calls can then be accepted in sequence using an entry family approach, provided by Ada, and a loop with an entry family index. A loop is used only if no loop already exists. This phase is explained in more detail in subsection 4.1. It is assumed here that each task in the program can be uniquely identified. The advantage of having more than one ATC is significant in multiprocessor systems. For example, having one ATC for each group of tasks that run on the same processor reduces the amount of communications between processors.

The second phase is to control nondeterministic selections inside

Ada tasks. This is done by extending each task T into T' such that the semantics of T and T' are the same. Basically, each task keeps track of its local rendezvous sequence and enables only the rendezvous at the top of its local rendezvous sequence and disables all other alternative rendezvous. The index to the local rendezvous sequence will then be updated, and the process will be repeated again until all rendezvous occur. The entry call for the enabled rendezvous is assumed to be available from the first phase. If it is not, the task will be blocked until the appropriate entry call arrives. More details are given in subsection 4.2. These two phases work together to achieve the reproduction of Ada tasking programs.

The separation between the two phases of the approach is necessary because each phase solves a separate problem in the reproduction process. The advantages are to simplify the implementation of the two phases and to make each phase easier to understand. This is especially true at the stage where we compare the results of the program with the results of its reproduction. A failure in the reproduction of interprocess communications most probably points to a failure in the entry calls control phase.

4.1 Entry calls control

We will first explain Phase One in general and apply it to one entry. Later in this section, we will explain how to expand it to include more than one entry. An example of the two phases is given in section five.

Figure 2 shows a task K that contains an entry $WRITE$. It also shows that there are three entry calls to the $WRITE$ entry: x , y , and z (where x is at the top of the entry's queue). Note that in this figure, we are using symbols that are similar to the one given in reference [3]. We are also temporarily using the calling's task identity as the entry's call identity, i.e., we are using the letter x as a task name and as an entry call identification (similarly y and z).

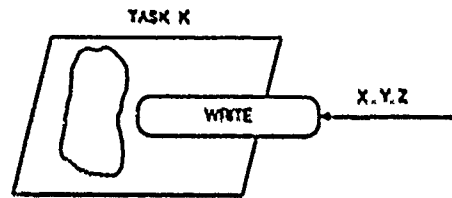


Figure 2. A task with one entry.

The purpose of phase one, in this example, is to insure that the $WRITE$'s entry queue is in the predetermined order at the reproduction execution, which is (x,y,z) . Figure 3 depicts how Phase One handles this problem. Each entry call to the $WRITE$ entry is extended into two calls.

The first call is to an ATC that handles task K . This first call is called the sign-in call. The purpose of this call is to get an entry call sequence number, which corresponds to the order of the call at the $WRITE$'s entry queue. Tasks x , y , and z can call the Ada task controller in any order, but they will always get the same entry call sequence number, e.g., the call issued by task z to the ATC will return an entry call sequence number of three.

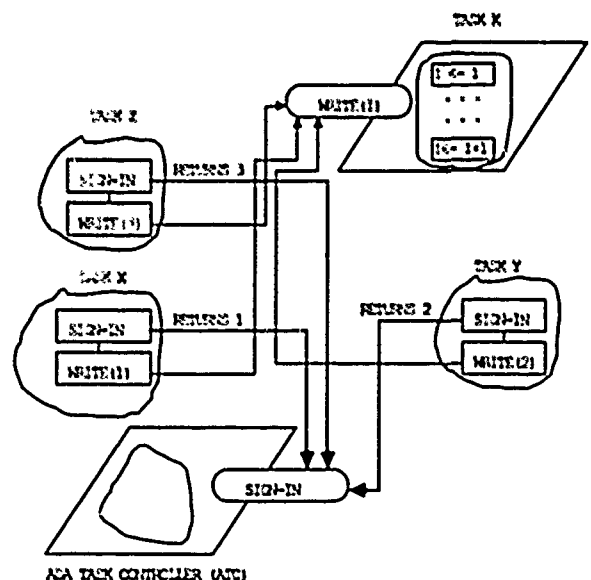


Figure 3. Sign-in calls in phase one.

The second call uses the entry call sequence number to call the original entry using an entry family approach. Task τ issues the entry call WRITE(3). These two entry calls are similar to the two-stage sign-in process suggested for explicit scheduling in reference [12].

To preserve the order of calls, the WRITE entry accepts one call at a time using an entry family and a loop with an entry family index (I), which is initialized to one and incremented by one every time the entry is involved in a rendezvous. Note, the calls to WRITE(I) can only be accepted in the following order: WRITE(1), WRITE(2), and WRITE(3). This preserves the original (x,y,z) entry calls sequence. A loop is required if no loop already exists, and an entry family index is required as shown in Figure 3. The loop stops when no more calls are issued to the entry. In summary, the number of iterations an entry goes through is equal to the number of calls (rendezvous) in which the entry is involved.

The ATC that handles task K should have an access to the predetermined entry calls sequence of the WRITE's entry. This enables it to assign an entry call sequence number to each call it receives. The ATC recognizes a call by the identity of the calling task, which should be passed to the ATC as an input parameter by the sign-in call. In the approach, it is part of the preprocessor to build an entry calls sequence for each entry within a task from the task's local rendezvous sequence. It is also part of the preprocessor to make these sequences accessible to the appropriate ATC.

Let us now assume that there are two entries in task K, a READ and a WRITE. In this case, two entry call sequences are built from a task's local rendezvous sequence, one for each entry. We will also use two entry family indices. An ATC in this case treats each sequence independently of the other. The ATC assigns an entry call sequence number according to the calling's task name and the called's entry name. These two names are passed as parameters by the sign-in call (refer to the READ procedure in Appendix II). This means that the above discussion regarding

entry extension applies to multiple entries regardless of the number of entries within a task. The only difference is that an ATC has to control one more entry calls sequence for each additional entry. The conclusion is that each entry is extended into an entry family with an initial entry family index of one and a limit of the number of entry calls to an entry.

It is clear from the above discussion that the original Ada source program should be extended to accommodate new Ada code for the purpose of reproduction. A summary of the extension procedure used in this phase is as follows:

1. Each entry call is extended into two calls, a sign-in call and a call to the original entry using an entry family approach.
2. Each accept is extended to an entry family, a loop (if required), and an entry family index.
3. An ATC is created for each task (or a group of tasks) to handle entry calls sequences, which are provided by the next step of this list.
4. A sequence of entry calls is built from a local rendezvous sequence of a task for each entry it contains.

These extensions are problem independent. The approach assumes that these extensions are part of the preprocessor. Refer to example one for more about these extensions.

4.2 Nondeterministic Selections Control

In Phase One, we tried to insure that entry calls to every task arrive in a predetermined order. This phase insures that a task selects a predetermined sequence of selections from a set of different alternatives available to it. We mentioned earlier that the selective wait statement is what makes an Ada task select a different selection from the same set of alternatives every time it is executed (local nondeterminism). The approach handles local nondeterminism

by handling the selective wait statement. We assume that no rendezvous nesting exists within the selective wait statement. The other types of select statement, namely the conditional entry call and the timed entry call, are handled by the approach. Although approach similar to the one given in reference [21] can be adopted. The approach also does not handle the COUNT attribute and shared variables. An approach for handling these two attributes is given in reference [21].

In this phase, each task follows its own local rendezvous sequence. This is done in two steps: Step One is to extend each task in an Ada source program to include a list of its own local rendezvous sequence; Step Two is to include a condition in each entry's When-clause that matches its own name with the entry name at the top of the task's local rendezvous sequence to which it belongs. The top of a local rendezvous sequence is determined by using a rendezvous index which is initialized to one and incremented by one after every rendezvous that occurs within the same loop (refer to RW task in Appendix II).

Conditions serve as guards to entries [14]. In the set of conditions, only one condition is always true. The true condition allows the rendezvous at the top of the local rendezvous sequence to occur. The false conditions prevent any other rendezvous to occur. The When-clause always signals the entry that the task should be involved in next. The other partner of the rendezvous, which is viewed as an entry call by the destination task, is provided by Phase One of the approach. After each rendezvous, an index that points to the next rendezvous in the local rendezvous sequence is incremented by one.

To see how the two phases work together, assume that a task is used with two entries, READ and WRITE. A local rendezvous sequence of this task is ($\langle 1, x, \text{READ} \rangle$, $\langle 2, y, \text{WRITE} \rangle$, $\langle 3, z, \text{READ} \rangle$). This three-rendezvous list contains three sub-lists. Each sublist represents a rendezvous. Each sublist contains three entities: a rendezvous sequence number, a name of

a calling task, and a called entry name. Assume also that the entry calls coming from tasks x, y, and z arrive in a predetermined order by Phase One. The task has a loop that iterate three times and involves in three rendezvous then terminates (see Figure 4).

When executing task G (Figure 4) and during the first iteration, the READ and WRITE alternatives are available. The READ's When-clause becomes true because it was involved in the first rendezvous. The WRITE's When-clause becomes false. So the READ entry is selected for the first rendezvous. The other partner of the rendezvous is task x. By assumption, the entry call from task x to the READ entry is at the top of the READ's entry queue. The two partners of the first rendezvous now match and the rendezvous occur. The two other rendezvous are reproduced in the same way.

This phase requires some extensions to an Ada source program. The first extension is to make each task in a program access its own local rendezvous sequence. A local rendezvous sequence is represented as an array of entry names. A rendezvous index is needed to point to the next rendezvous in the sequence. The last rendezvous occurs at the last iteration of the selective wait statement.

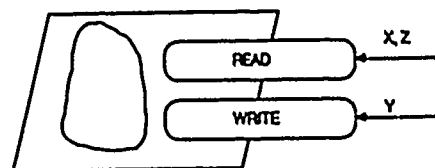


Figure 4. A task with two entries.

The second extension is to include in each entry's When-clause a condition that matches the entry's own name with the name of the entry at the top of the task's local rendezvous sequence. The approach assumes that these two extensions are part of the preprocessor. The next

section explains the reproduction process of an extended Ada program.

5. A COMPLETE EXAMPLE

An Ada program is listed in Appendix I [12]. The program is a controller for a shared resource that allows multiple readers at the same time and only one writer at a time. In this example, we plan an execution scenario of the program and then determine how this scenario is specified. We also explain the needed extensions to the original program.

EXAMPLE 1:

Using the Ada code in Appendix I, assume that there are four demanding tasks C1, C2, C3, and C4 that use the RESOURCE package. Further assume that C1 and C2 called RW for reading the resource where C1 called before C2. Task C3 called for writing while C1 and C2 were still in the process of reading. Task C4 called for reading after C3 had finished writing. Because task RW has an infinite loop, assume here that the number of readers and writers are finite and it will terminate.

To specify the above scenario, we need to determine a local rendezvous sequence that represents the above path. One possible local rendezvous sequence is:

```
LRS1 := ( <1,C1,START_READ>,
          <2,C1,END_READ>,    <3,C2,START_READ>,
          <4,C2,END_READ>,    <5,C3,START_WRITE>,
          <6,C3,END_WRITE>,  <7,C4,START_READ>,
          <8,C4,END_READ> )
```

Recall that each rendezvous is represented as:

<Rendezvous sequence number, Calling task identity, Entry name>.

Rendezvous 2 and 3 can be exchanged to get another local rendezvous sequence that still represents the same path. The above local rendezvous sequence is a path specification for the program in Appendix I with the execution scenario explained above. LRS1 is the content of the path specification file (see Figure 1). Note that tasks C1, C2, C3, and C4 are

assumed to be demanding tasks (refer to the end of Section 3 for the definition of a demanding task), and as a result, no local rendezvous sequences are specified for them.

When this path specification is read by the preprocessor, it builds a sequence of entry calls for each entry in the RW task. The preprocessor builds the following entry calls sequences.

```
START_READ_SEQ := (C1, C2, C4)
END_READ_SEQ := (C1, C2, C4)
START_WRITE_SEQ := (C3)
END_WRITE_SEQ := (C3)
```

This set of entry calls sequences are then built into an ATC for task RW (refer to RW_C task in Appendix II). The preprocessor creates the RW_C as an Ada task controller for RW task and adds it to the original Ada source program. Note that it should be specified that an ATC be built for the RW task in the preprocessor command file. The preprocessor also extends an Ada program according to the extension procedures of the two phases given in subsections 4.1 and 4.2. Appendix II lists the program after it has been extended by the preprocessor.

There are two packages in Appendix II: the RESOURCE package and a CONTROLLER package. The RESOURCE package is extended to accommodate new Ada code. A symbol at the end of a line indicates how much a line is extended. The symbol "--Q" indicates that the line is added completely to the original program. The symbol "--&" indicates that some extension occurred in the line. Note that the CONTROLLER package is completely added so there is no need for using any symbols.

Note how each call to the RW task is extended in the READ and WRITE procedures as a result of the entry calls control phase. The sign-in call to the controller (RW_C) includes the caller identification, the requested entry name, and a dummy parameter to return an entry sequence number. The original call is extended to include the entry call sequence number (ENTRY_CALL_SEQ_NO). Note also how each entry is extended in the RW task. Each entry is extended to a family of entries and has its own

family of entry index. These extensions are part of Phase One; the rest of the extensions in the RW task are part of Phase two. However, the CONTROLLER package is a result of the extensions in Phase One.

Note also how each entry's When-clause was extended to include an additional condition and how the while loop keeps track of the number of rendezvous for reproduction using the rendezvous index (NEXT_R). Note also how the local rendezvous sequence (LOCAL_REND_SEQ) is represented. These extensions are a result of the nondeterministic selections control phase.

As a result of these extensions, the specification of the RESOURCE package, and the RW task were extended. The program given in Appendix II is a deterministic version of the original program in Appendix I, and it will always follow the same path (specified by LRS1) every time it is executed.

6. CONCLUSIONS

Reproduction of Ada tasking programs is a problem that must be dealt with in cyclic debugging of Ada programs. The method of avoiding the problem by building a debugger that has the ability to discover and locate errors is inadequate because this method mixes the testing and debugging phases, is complex, and is expensive. The method of extending a nondeterministic program into a deterministic one is adequate because it is easy to understand and to implement; however, the problem is difficult in Ada because of the asymmetric naming convention to the way Ada handles entry queues. The approach distinguishes between two types of nondeterminism: global nondeterminism and local nondeterminism. It handles each type separately. The approach extends a nondeterministic Ada tasking program into a deterministic one. This is done in two phases: Phase One handles global nondeterminism, and Phase Two handles local nondeterminism. Global nondeterminism is handled by using one Ada task controller per task to control the arrival sequence of calls to a destination task. Having more

than one controller eliminates the need for a master controller that can be a bottleneck to the reproduction process. Local nondeterminism is handled by restricting the number of open alternatives in a selective wait statement using a When-clause as a guard to each alternative. The method is easy to understand and to implement.

APPENDIX I

package body RESOURCE is
S : SHARED_DATA := -- The shared data

task RW is
 entry START_READ;
 entry END_READ;
 entry START_WRITE;
 entry END_WRITE;
end RW;

task body RW is
 NO_READERS: NATURAL := 0;
 WRITER_PRESENT: BOOLEAN := FALSE;
 begin
 loop
 select
 when not WRITER_PRESENT =>
 accept START_READ;
 NO_READERS := NO_READERS + 1;
 or
 accept END_READ;
 NO_READERS := NO_READERS - 1;
 or
 when not WRITER_PRESENT AND
 NO_READERS = 0 =>
 accept START_WRITE;
 WRITER_PRESENT := TRUE;
 or
 accept END_WRITE;
 WRITER_PRESENT := FALSE;
 end select;
 end loop;
 end RW;

procedure READ (X:out SHARED_DATA) is
 begin
 RW.START_READ;
 X := S;
 RW.END_READ;
 end READ;

procedure WRITE(X : in SHARED_DATA) is
 begin
 RW.START_WRITE;
 S := X;
 RW.END_WRITE;
 end WRITE;
end RESOURCE;

APPENDIX II

```

with CONTROLLER;                                --0
package RESOURCE is                               --0
use CONTROLLER;                                  --0
  type SHARED_DATA is ...;
  procedure READ(CALLER_ID:in                    --&
                 CALLER_NAME:                    --&
                 X : out SHARED_DATA);
end RESOURCE;

package body RESOURCE is
S : SHARED_DATA := -- The shared data

task RW is
  entry START_READ(REND_INDEX);                  --&
  entry END_READ(REND_INDEX);                    --&
  entry START_WRITE(REND_INDEX);                 --&
  entry END_WRITE(REND_INDEX);                   --&
end RW;

task body RW is
LOCAL_REND_SEQ:RENDEZVOUS_LIST;                  --0
NEXT_R : REND_INDEX :=1;                         --0
SR,ER,SW,EW:REND_INDEX := 1;                    --0
LOCAL_REND_SEQ (1 .. 8) :=
(START_READ, END_READ,                          --0
 START_READ, END_READ,                          --0
 START_WRITE, END_WRITE,                        --0
 START_READ, END_READ);                         --0
NO_READERS: NATURAL := 0;
WRITER_PRESENT: BOOLEAN := FALSE;

begin
while NEXT_R <=
  RW_REND_INDEX_LIMIT --0
  loop
  select
  when not WRITER_PRESENT and
    LOCAL_REND_SEQ(NEXT_R) =
      START_READ => --0
    accept START_READ(SR); --&
  NO_READERS := NO_READERS + 1;
  SR := SR + 1; --0
  or
  when LOCAL_REND_SEQ(NEXT_R) =
    END_READ=>accept END_READ(ER); --&
  NO_READERS := NO_READERS - 1;
  ER := ER + 1; --0
  or
  when not WRITER_PRESENT and
    NO_READERS = 0 and
    LOCAL_REND_SEQ(NEXT_R) =
      START_WRITE => --0
    accept START_WRITE(SW); --&
  WRITER_PRESENT := TRUE;
  SW := SW + 1; --0
  or
  when LOCAL_REND_SEQ(NEXT_R)=
    END_WRITE => --0
    accept END_WRITE(EW); --&
  WRITER_PRESENT := FALSE;
  EW := EW + 1; --0
  end select;
  NEXT_R := NEXT_R + 1;
  end loop;
end RW;

procedure READ(CALLER_ID :in
CALLER_NAME; X :out SHARED_DATA) is
begin
  RW.C.SIGN_IN(CALLER_ID,
START_READ,ENTRY_CALL_SEQ_NO); --0
  RW.START_READ(
ENTRY_CALL_SEQ_NO); --&
  X := S;
  RW.C.SIGN_IN(CALLER_ID, END_READ,
ENTRY_CALL_SEQ_NO); --0
  RW.END_READ(ENTRY_CALL_SEQ_NO); --&
end READ;

procedure WRITE(CALLER_ID :in
CALLER_NAME; X :in SHARED_DATA) is
begin
  RW.C.SIGN_IN(CALLER_ID, START_WRITE,
ENTRY_CALL_SEQ_NO); --0
  RW.START_WRITE(
ENTRY_CALL_SEQ_NO); --&
  S := X;
  RW.C.SIGN_IN(CALLER_ID, END_WRITE,
ENTRY_CALL_SEQ_NO); --0
  RW.END_WRITE(ENTRY_CALL_SEQ_NO); --&
end WRITE;
end RESOURCE;

package CONTROLLER is
type ENTRY_NAME is (START_READ,
END_READ,START_WRITE, END_WRITE);

RW_REND_INDEX_LIMIT : constant := 8;

type REND_INDEX is POSITIVE;

type RENDEZVOUS_LIST is array
REND_INDEX) of ENTRY_NAME;

type CALLER_NAME is (C1, C2, C3, C4);

type ENTRY_QUEUE is array(REND_INDEX)
of CALLER_NAME;

procedure SEARCH( SEQUENCE : in out
ENTRY_QUEUE;
ID : in CALLER_NAME;
OUT_NO : out REND_INDEX);
end CONTROLLER;

```

```

package body CONTROLLER is

task RW_C is
entry SIGN_IN (CALLER_ID : in
                CALLER_NAME;
                ENTRY_REQ : in ENTRY_NAME;
                CALL_SEQ_NO : out REND_INDEX);
end RW_C;

task body RW_C is

CALL_INDEX : INTEGER := 0;
START_READ_SEQ, END_READ_SEQ :
    ENTRY_QUEUE;
START_WRITE_SEQ, END_WRITE_SEQ :
    ENTRY_QUEUE;
START_READ_SEQ (1 .. 3) := (C1,C2,C4);
END_READ_SEQ (1 .. 3) := (C1,C2,C4);
START_WRITE_SEQ (1) := (C3);
END_WRITE_SEQ (1) := (C3);
begin
loop
when CALL_INDEX <=
RW_REND_INDEX_LIMIT ->
accept SIGN_IN (CALLER_ID : in
                CALLER_NAME;
                ENTRY_REQ : in ENTRY_NAME;
                CALL_SEQ_NO : out REND_INDEX) do
case ENTRY_REQ is
when START_READ ->
SEARCH(START_READ_SEQ;
      CALLER_ID;
      CALL_SEQ_NO);
when END_READ ->
SEARCH(END_READ_SEQ;
      CALLER_ID;
      CALL_SEQ_NO);
when START_WRITE ->
SEARCH(START_WRITE_SEQ;
      CALLER_ID;
      CALL_SEQ_NO);
when END_WRITE ->
SEARCH(END_WRITE_SEQ;
      CALLER_ID;
      CALL_SEQ_NO);
when others -> null;
end case;
end SIGN_IN;
CALL_INDEX := CALL_INDEX + 1;
end loop;
end RW_C;

procedure SEARCH( SEQUENCE : in out
                  ENTRY_QUEUE;
                  ID : in CALLER_NAME;
                  OUT_NO : out REND_INDEX) is
begin
INDEX := 1;
while INDEX <= SEQUENCE'LENGTH
loop
if SEQUENCE(INDEX) = ID then
OUT_NO := INDEX;
SEQUENCE(INDEX) := null;

```

```

exit;
else
INDEX := INDEX + 1;
end if;
end loop;
end SEARCH;
end CONTROLLER;

```

REFERENCES

- [1] F. Baiardi, N.D. Francesco, G. Vaglini, "Development of a Debugger for a Concurrent Language," IEEE Trans. on Software Engineering, VOL. SE-12, NO. 4, April 1986, pp. 547-553.
- [2] H. Barringer, I. Mearns, "A Proof System for Ada Tasks," The Computer Journal, Vol. 29, NO. 5, 1986, pp. 404-415.
- [3] G. Booch, "Software Engineering With Ada," The Benjamin/Cummings Company, California, 1983.
- [4] Per Brinch Hansen, "Reproducible Testing of Monitors," Software-Practice and Exper., Vol. 8, 1978, pp. 721-729.
- [5] T. Elrad, "A Practical Software Development for Dynamic Testing of Distributed Programs," Proceedings of the 1984 International Conference on Parallel Processing, August 1984.
- [6] T. Elrad, "Data Dependencies Within Distributed Programs," Proceedings of the Hawaii International Conference on System Sciences, January 2, 1985.
- [7] T. Elrad and N. Francez, "Decomposition of Distributed Programs Into Communication-closed Layers," Science of Computer Programming 2, North-Holland, 1982, pp. 155-173.
- [8] T. Elrad, F. Maymir-Ducharme, "Race Control for the Validation and Verification of Ada Multitasking Programs," Proceedings of the Sixth Annual National Conference on Ada Technology, May 14-17, 1988.
- [9] R. G. Fainter and T.E. Lindquist, "Debugging Tasked Ada

Programs," Proceedings of the Ada Applications for the NASA Space Station Conference, in R.L. Bown(ed.), June 1986, pp. B.1.1.1-23.

- [10] N. Francoz, C.A.R. Hoare, D. J. Lehmann, W. P. DE Roover, "Semantics of Nondeterminism, Concurrency, and Communication," Journal of Computer and System Sciences 19, 1979, pp. 290-308.
- [11] H. Garcia-Molina, F. Germano, W. Kohler, "Debugging a Distributed Computing System," IEEE Trans. on Software Engineering, Vol. SE-10, No. 2, March 1984, pp. 210-219.
- [12] N. Gehani, "Ada Concurrent programming," Prentice-Hall, New Jersey, 1984.
- [13] D. Helmbold, D. Luckham, "Debugging Ada Tasking Programs" IEEE Software, March 1985, pp. 47-57.
- [14] C.A.R. Hoare, "Communicating Sequential Processes," CACM, Vol. 21, NO. 8, August 1978, pp. 666-677.
- [15] S. Katz, D. Peled, "Interleaving Set Temporal Logic," Proc. of the Sixth Annual ACM Symposium on Principles of Distributed Computing, August 1987, pp. 178-190.
- [16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Comm. of ACM, Vol. 21, NO. 7, July 1978, pp. 558-565.
- [17] T.J. Leblanc, J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," IEEE Trans. on Computers, Vol. C-36, NO. 4, April 1987, pp. 471-482.
- [18] B.P. Miller, "A Mechanism for Efficient Debugging of Parallel Programs," SIGPLAN NOTICES, Vol. 23, NO. 7, July 1988, pp. 135-144.
- [19] A. Pnueli, "The Temporal Semantics of Concurrent Programs," Theoretical Computer Science, Vol. 13, 1981, pp.

45-60.

- [20] J.M. Stone, "Debugging Concurrent Processes: A Case Study," SIGPLAN NOTICES, Vol. 23, NO. 7, July 1988, pp.145-153.
- [21] K.C. Tai, E.E. Obaid, "Reproducible Testing of Ada Tasking Programs," Proc. IEEE-CS Second Inter. Conf. on Ada Applications and Environments (1986), pp. 69-79.
- [22] K.C. Tai, S. Ahuja, "Reproducible Testing of Communication Software," Proc. of IEEE COMPSAC 87, Oct. 1987, pp. 331-337.



Mamdouh Najjar received the B.S. and M.S. degrees in computer science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in 1982 and 1986, respectively.

He is currently a full-time Ph.D. student at Illinois Institute of Technology, Chicago, Illinois. His research interests include, concurrent programming, distributed systems, and distributed testing and debugging.



Tzilla Elrad received an M.S. degree in computer science from Syracuse University, N.Y. and a Ph.D. in computer science from the Technion in Israel, in 1978 and 1981, respectively.

She is an assistant professor of computer science at Illinois Institute of Technology. Her main interests are in concurrent and distributed languages, concurrent programming for real-time applications and the use of Ada for such systems. She is the chair of Chicago SIGAda. Her BITNET address is: CSELRAD@IITVAX.

Problems Encountered in Learning Object Oriented Design Using Ada

Greg Carlson

Undergraduate -- St. Cloud State University
St. Cloud, MN 56310

Abstract

Developing an understanding of object oriented programming in Ada presents challenges for the experienced programmer. The project presented is a text adventure game using object oriented methodology and Ada. We can improve the learning process involved in these topics by examining the barriers encountered by a student while designing this project.

Introduction

Learning a new programming method can be difficult. As a student learning object oriented programming this challenge was faced. The project presented was to learn object oriented design and Ada by developing a text adventure game. The goal of this project was to become familiar with the Ada programming language and to gain an understanding of object oriented programming. The major barrier encountered was to break out of the rooted procedural style of programming and adopt object oriented style. Due to inexperience in the object oriented technique, problems were encountered throughout the stages of development. Dealing with these problems led to a greater level of understanding. This paper describes important issues involved in the design and implementation of a major programming task using object oriented programming in the Ada environment.

Project Description

The project was presented in the first quarter of a three quarter software engineering class. An open specification of the problem was given, allowing the programmers to expand on the design. The original concept was to create a dungeon or maze made up of interconnected rooms. The goal of the game is to collect treasure found in these rooms and rescue the kidnapped princess. The final program included the rooms and objects, but also added creatures and doors.

The player moves from room to room collecting treasure. When a room is entered, a description of the room is given along with the names of the creatures and treasure it contains. Treasure within the rooms carry point values that are added to the player's score when picked up and subtracted when dropped. There are four subclasses of items: general treasure (worth varying point values), Heavy treasure (worth negative point values), weapons (needed to kill creatures), and keys (to lock and unlock doors).

Commands that can be used by the player include operations on treasure, creatures, doors, and other miscellaneous commands. The player takes, drops, and examines treasure. When a player is in the same room as a creature the creature is talked to, examined, and attacked. A creature is killed when the player is carrying a weapon and is stronger than the creature. Doors are locked, unlocked, opened, and closed. Locking and unlocking can only be done when the player is carrying a key. Other commands allow a user to move from room to room, check to see what objects are being carried, view a help screen, and quit the game.

Project Development

Identifying the Objects.

Objects are the entities in the problem that act as nouns.[1] A good understanding of objects is obtained by imagining them as being people. Each person has traits that are represented by fields in the object.

There were five objects in the initial project design: rooms, corridors, doors, items, and creatures. The object named creatures, when first defined, contained the fields and operations for the player and the monsters. After examining the combination, it was found that the player and creature were two separate types with little overlap in operations. The creature object was then split into the objects creatures (monsters) and player. Realizing that the player and monster objects were separate, led to an understanding of cohesion in objects.[2] Another refinement occurred between the rooms object and the corridors object. Analyzing the description of the two objects forced the realization that the two were essentially the same. By combining the two objects an increased understanding of well defined abstract data types evolved. After refinement, the object identification was complete. The final identification of the objects included an object for each of the following: rooms and corridors, doors, items, creatures and player.

Identifying the Operations.

The operations in the problem are the entities that act as verbs.[1] Operations allow fields within an object to be accessed or changed. Operations must be complete. In order for an object to be complete, the operations must allow access to all of the visible fields in the object. Using the analogy of objects as people, the operations can be thought of as interactions between people. One person may request information from another or may attempt to change that person (sometimes a person will talk to themselves).

One of the major problems encountered in designing this project was identifying the operations. At first the operations seemed logical and complete but when establishing the interface, the operations were found to be

incomplete. The initial identification of the operations did not take into consideration the strict enforcement of information hiding. When it came time to establish the interface, it was realized that extra operations were needed to manipulate the objects. It was at this point that an understanding of how to define complete operations was developed.

Establishing the Visibility.

The visibility is determined by the relationships between the objects. One object is visible to another if it is used by the other object. Person A is visible to Person B if B asks A questions.

When establishing the visibility, two problems were encountered: over-dependent objects and codependent objects. The over-dependence of the objects was shown by the dependency graph. The number of connections going from a major object to other major objects was large. Codependency is caused when two objects must be visible to each other. Ada does not permit codependency. When using the *with* statement to establish visibility of an object, the packages being accessed must be compiled before the accessing package can be compiled. If two objects must be visible to each other then it would be impossible to compile them. In the project, both the over-dependency and the codependency was caused by poor design. Major objects would access each other in order to perform commands. To remove this problem a hierarchy of objects was instituted (see figure 1). The upper level of the hierarchy controlled the command flow by accessing the major objects. This solution to this problem brought to light the importance of structure in defining object interaction.

Establishing the Interface.

The interface defines what other objects are allowed to access in a particular object. No problems arose from using the module specifications to establish the interface. Very explicit rules of information hiding were required in the project specification. By requiring the use of private types in package specifications, information hiding was enforced.

Conclusion

Following are the five major areas in which learning difficulties occurred:

1. Defining objects that follow the rules of abstract data types.
2. Defining complete operations.
3. Structuring the object organization.
4. Implementing communication between objects.

Since the lab was designed so that the object oriented methodology would be learned by doing, a solution to each problem marked a landmark in the learning process. The use of the Ada programming language facilitated the object oriented design due to the implementation of packages and private types. An introduction to the object oriented programming method gave a good understanding of the process. When the object oriented programming method was applied, a complete understanding of the method was gained. It is hoped that, through examining the process involved in learning object oriented programming in Ada, the understanding of these concepts and how they are presented can improve.

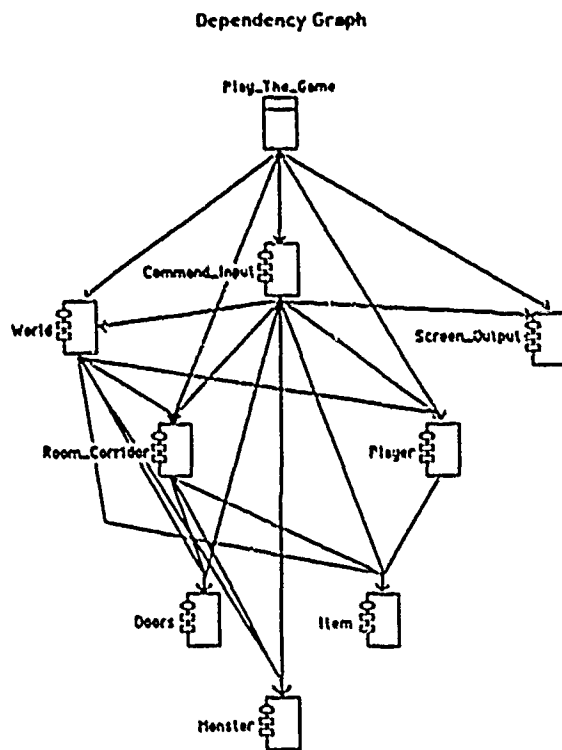


Figure 1

Implementing the Objects.

Objects are implemented by transferring the object oriented representation into Ada code. A design barrier involving message passing was introduced while implementing the objects. The problem arose when trying to implement how a player or a room could possess treasure. Information hiding does not allow either the room/corridor object or the player object to actually access the Items implementation. The problem was solved by the implementation of access types. The access type identifies the item that is being passed from object to object. The understanding of how to implement objects while following the rules of information hiding was improved by overcoming this difficulty.

References

1. G. Booch, Software Engineering with Ada, Second Edition, Benjamin/Cummings Publishing Company, Menlo Park, California, 1987.
2. D. Embley and S. Woodfield, "Assessing the Quality of Abstract Data Types Written in Ada," Proceedings of the 10th International Conference on Software Engineering, pp. 144-153, IEEE, Singapore, April, 1988.
3. D. Lamb, Software Engineering: Planning for Change, Prentice Hall, Englewood Cliffs, New Jersey, 1988.



Greg Carlson is an undergraduate student at St. Cloud State University, St. Cloud State, MN. Interests include software management, neural networks, and computer graphics.

QUEST FOR USABILITY IN ADA GENERICS

Kevin A. Minder *

Trenton State College
Millwood Lakes CN 4700
Trenton, NJ 08650-4700

Abstract. One of Ada's most powerful features is the ability to develop general algorithms to work in a variety of different situations. The widespread use of generic units will help Ada become the high level design language that it is intended to be. However, the development of generic units can be a difficult process and there are many things to be considered. Most importantly, care must be taken to ensure that the generic units are designed in such a manner as to encourage their use. This paper will give practical examples of a generic unit written in ways that will both encourage and discourage its use. These examples will show how careless design of generic units can lead to the impracticality and failure of generics as a powerful feature of Ada.

While working in the software engineering industry as an intern I had an opportunity to be taught Ada, have access to a great deal of literature on Ada, and have the opportunity to spend time working with the language. Early on, many lessons were taught about what made Ada a different and useful language. By the end of my stay as an intern, the more powerful features of Ada had been revealed. The reasons for use and the methods of use were also well explained. One of the features that seemed to be the most powerful and beneficial to the overall purpose of the Ada language was generics.

I found that when a beginner is being taught about generics, the idea of total abstraction from data as the goal of generics was very strongly emphasized. In the same respect, examples were produced that led to very nice results that displayed a very high level of abstraction from data. However, it is only through handpicking these examples that an "easy" solution renders itself.

Later, upon continuing study and after working with many problems using generics for a variety of practical applications, I realized that my designs of many generic units led to cumbersome and hard to use solutions. In addition, and most importantly, I realized that by being awkward and hard to use these generic units defeated their own purpose.

This paper will provide a practical example of a generic unit that if not carefully designed will lead to code that would defeat the entire purpose of it being a generic. The generic unit used will be in the form of a generic Sort. Since total abstraction is usually stressed, in teaching generics and in the literature on generics, the first several examples of this generic unit will have complete abstraction from the information to be sorted and the data structure that the information is contained within. The method of sorting and therefore the body of any of the generic units are completely irrelevant to this discussion since the user of a generic unit should only be concerned with the specification of that unit.

One example of a specification for a generic unit that could be used to sort any type of data within any type of data structure is this specification.

generic

```
type Element_Type is private ;
type Key_Type      is private ;
type List_Type     is private ;
Number_Of_Elements : positive ;
```

```
with function "<"
  ( Key_1 : Key_Type ;
    Key_2 : Key_Type )
return Boolean ;
```

```
with function ">"
  ( Key_1 : Key_Type ;
    Key_2 : Key_Type )
return Boolean ;
```

```

with function Key
  ( Element: Element_Type )
  return Key_Type ;

with function Next_Element
  ( Element: Element_Type )
  return Element_Type ;

with procedure Replace
  ( Old_Element: in out Element_Type ;
    New_Element: in   Element_Type );

procedure Sort ( List: in out List_Type);

```

While this specification for a generic sort procedure may seem ridiculous, it is designed to be. It is intentionally designed to display, in the worst case, just how unusable a generic unit can become. Fortunately, if this example is re-thought much can be done to improve it. The most obvious negative point is that there are nine generic formal parameters, six of which are generic formal subprograms. It is likely that someone requiring a sort could write a non-generic sort in less time than required to instantiate this specific generic sort. This is probably true even if very good documentation was provided concerning the function of each generic formal parameter.

There are several quite obvious improvements that can be made to alleviate the problems that this example has with unnecessary and inhibiting complexity.

1. The overloaded inequality functions can be reduced from two functions (i.e. "<" and ">") to just one of the two function is always possible.
2. The generic formal function that returns the key of an element is not necessary. This operation can be done in conjunction with the now single generic subunit for inequality.

A specification taking these two changes into account might look like to following code.

generic

```

type Element_Type is private ;
type List_Type     is private ;
Number_Of_Elements : positive ;

```

```

with function "<="
  ( Element_1 : Element_Type ;
    Element_2 : Element_Type )
  return Boolean ;

with function Next_Element
  ( Element : Element_Type )
  return Element_Type ;

with procedure Replace
  ( Old_Element: in out Element_Type ;
    New_Element: in   Element_Type );

procedure Sort ( List: in out List_Type);

```

With this obvious adjustment the number of formal parameters has been reduced from nine to six with only three of these being generic formal subunits. This adjustment will reduce the overall complexity of using the generic unit somewhat, however, two of the three remaining generic formal subunits, Next_Element and Replace, will be very difficult for the user of a generic package to implement. This is due to the fact that the person using the package must have a good feel for what the designer of the generic package had in mind for these subprograms when writing the generic unit.

An example of what the designer might have had in mind when designing the above generic unit might be these descriptions.

1. Next_Element might be a generic formal subprogram that given a particular element in the external data structure would return just the data part of the next element in the structure. This alone would require a linear search.
2. Replace would replace each unsorted element with the sorted element corresponding to the unsorted elements position. This subprogram would also require a linear search.

It is obvious that the logic of these generic formal subprograms is more difficult and lengthy than they could optimally be. This problem arises because the design of the generic sort attempts to abstract completely away from the data structure that it will be sorting. Since the two difficult generic formal subunits provide methods of transferring the unsorted and sorted material to and from the generic

procedure, then the only way to simplify the problem is to find an easier method to perform these operations.

Traditionally, the way to make any algorithm easier to understand, is to use an algorithm that mimics an intuitive approach. In the case of a sort, a very intuitive approach is available. That is, to give all data to something and to expect to receive the data back in sorted order. This can be translated into copying all the elements in the external data structure into the internal data structure, sorting the internal data structure, and then copying the sorted elements out to the external data structure.

One elegant way of achieving this is to change the generic sort procedure into a generic package containing an interface task. This interface task can transfer an element from the calling unit to the generic package very easily. In this situation the generic package could look like this:

```
generic
  type Element_Type is private ;

  with function "<"
    ( Element_1 : Element_Type ;
      Element_2 : Element_Type )
    return Boolean ;

package Sort_Package is

  task Transfer is

    entry Send
      ( Element      : in Element_Type ;
        Last_Element : in Boolean ) ;

    entry Receive
      ( Element      : out Element_Type ;
        Last_Element : out Boolean ) ;

    -- Note: Two procedures must be
    -- included in the calling program to
    -- use this sort. To use the sort
    -- execute the first procedure
    -- described below, Then execute the
    -- second procedure described below.
    -- The element in the second
    -- procedure will be received in
    -- sorted order.

    -- The first procedure should make
    -- entries into the task at Send with
    -- the first through the next to the
    -- last elements in the external
```

```
-- data structure for the Element
-- parameter and a value false for
-- the Last_Item parameter. Finally,
-- an entry should be made into the
-- task at Send with the last value
-- in the external data structure for
-- the Element parameter and with a
-- value of true for the
-- Last_Element parameter.
```

```
-- The second procedure should make
-- consecutive entries into the
-- task at Receive expecting the
-- first through next to last sorted
-- elements from the Element
-- parameter and false for the
-- Last_Item parameter each time.
-- With the last entry into the
-- task at Receive the last sorted
-- item in the internal data
-- structure will be returned
-- in Element and the parameter
-- Last_Element will be true.
```

end Transfer ;

end Sort_Package ;

Notice that List_Type and Number_Of_Element parameters are not present. That is because in this case the generic package needs to know nothing about the external data structure because no references are made to it from within the generic package itself. All elements will be removed from the data structure and given to the sort package through the interface task. Two procedures must be written, by a programmer using the package, to interface with the tasks. All necessary references to the external data structure can be made within these external procedures. Examples of the external procedures used to sort an array through the use of an interface task might resemble these two procedures.

```
procedure Send
  ( List : in List_Type ) is
begin
  for Index in 1..List_Type'Last-1 loop
    Sort_Package.Sort.Send
      ( List ( Index ), false );
  end loop ;
  Sort_Package.Sort.Send
    ( List ( List_Type'Last ), true );
end Send;
```

```
procedure Receive
  ( List : out List_Type ) is
  Last_Item : Boolean ;
begin
  for Index in List_Type'Range loop
```

```

Sort_Package. Sort. Receive
  ( List ( Index ), Last_Item );
end loop ;
end Receive;

```

These two procedures follow the intuitive method of performing a sort very closely and they are very simple to implement. It is important to note that these procedures are not included as generic formal subprograms. Therefore, the generic package specification should contain documentation explicitly stating that two procedures similar to the above must be used with the generic package and it should provide specific directions relating to operation of each of these procedures. This solution, using tasking, seems to be simple and effective enough so that anyone with a basic understanding of tasking could have the generic package working very quickly. However, if consideration is given to those who do not have a basic knowledge of tasking or to those projects which do not allow tasking, an alternate solution must be developed.

If the use of tasking is restricted, then any solution that may be developed is likely to suffer the same faults as did the previous examples. Consequently, to create an effective solution, a different approach to generics must be taken. This new approach will start at the beginning with reconsidering the general principle behind generics. Basically, the question that must be answered is, "How generic should generics be?" Generally, teaching and literature suggest that generics should provide abstraction from data. The previous examples provided complete abstraction from both the external data structure and the data to be sorted, but at a price.

Maintaining this extra capability of the generic unit requires the addition of several generic formal parameters, or some other method of interface between the generic unit and the calling unit. Keeping this in mind the question, "Is complete abstraction from the external data structure necessary or even advisable?", must be asked.

In the above case of the sort, abstraction from only the data to be sorted seems to be enough. Certainly, individual generic units will have to be developed to work with different types of list data structures if a complete

solution is to be added to any library. In particular a slightly different generic unit would have to be developed for arrays, linked lists, and other types of data structures. This is a small price to pay for having a generic unit that is simple and encourages its use. For example, the generic unit for sorting arrays with elements of any abstract type could be as simple as this generic.

generic

```

type Element_Type is private ;

with function "<"
  ( Element_1 : Element_Type ;
    Element_2 : Element_Type )
return Boolean ;

```

procedure Sort (List: in out List_Type);

In this case no method of transferring data to and from the generic unit is necessary since the external data structure is visible to the generic unit. Therefore, the array can be manipulated directly by the generic. This is clearly the simplest way to implement a generic sort providing that the data to be sorted is contained within an array.

The generic sorting of linked lists is not quite as simple. This is due to the fact that there is no standard format for linked list. However, if during the development of a particular application a sorted linked list is required, looking at the specification of the generic linked list sort and forming the linked list as prescribed by the generic unit will be a simple task. In any case following the form of the linked list imposed by the generic for the sorting of a linked list will be much easier than developing several confusing generic formal subunits required to use a completely abstracted generic. On the other hand, a complete solution might be to contain the generic sort within a generic package for linked list operations in which the design of the linked list would be known before hand. An example of a specification for a generic sort of a linked list might look similar to this:

generic

```

type Element_Type is private ;

with function "<"
  ( Element : Element_Type ;

```

```

    Element : Element_Type )
return Boolean ;

```

```

-- Individual nodes of the linked list
-- must take the form
--   record
--     Element : Element_Type ;
--     Next    : List_Type ;
--   end record ;
-- where Element_Type can be any type
-- excluding task types and constants.
-- List_Type must be a pointer to this
-- record.

```

```

procedure Sort ( List: in out List_Type);

```

The different approach taken to design these last two generics is a very significant one. If an attempt is made to make generic units too generic, they may become quite difficult to use. However, by narrowing the scope of generic units so that it abstracts from only certain elements will produce results that are much more likely to facilitate use.

Two different approaches to generics have been given here. For each approach to designing a particular generic unit examples have been given of ways to implement each approach. The basic principles behind making each example more usable are applicable to the development of any generic unit, and they are:

1. The number of generic formal parameters should be limited to the smallest number possible.
2. Each remaining generic formal subunit should be as easy as possible to understand and code.
3. If the generic unit designed under the principle of total abstraction from data produces hard to use generic formal subprograms, redesign the generic unit with decreased generic abstraction.
4. Lastly, of course document every aspect of the necessary generic formal parameters. In addition, as with the tasking example, explicitly document any necessary elements that do not appear in the generic specification. Provide examples if necessary.

In short, the designer of a generic must insure that the time spent, by a programmer, to understand how to use a generic unit must be substantially less than the time required by that programmer to implement a non-generic solution to the problem. Methods such as were described in this paper will ensure that this requirement is met.

The purpose of generics is eventually to provide a large library of re-usable code. If this goal is to be realized, then great care must be taken to ensure that each generic unit placed in these libraries is easy enough to use so that people will want to use them. If these precautions are not taken, one of the main goals of Ada may not become a reality.

Kevin A. Minder
69 South Locust Avenue
Marlton, NJ 08053

I was born in Wurtzberg, Germany and grew up in several states across the United States. I attended Cherokee High School, in Marlton, New Jersey. I am currently a junior Computer Science Major at Trenton State College, under a Garden State Distinguished Scholar Scholarship and Trenton State College Alumni Scholarship.



* Funding to support presentation of this paper was provided in part by a grant from Mobil Oil Co.

DESIGN CONSIDERATIONS AFFECTING IMPLEMENTATION OF BYZANTINE AGREEMENT PROTOCOLS IN ADA

Shoshana Hartman

Southeastern Massachusetts University
North Dartmouth, Mass. 02747

Abstract. Byzantine Agreement Protocols involve the execution of consensus algorithms to agree on the value sent by a transmitter in a distributed processing system in which some of the processors may fail in arbitrary and possibly malicious ways. This paper presents design considerations involved in an attempt to use these protocols within ADA intertask communication as a mechanism to support the design of fault tolerant systems.

1. INTRODUCTION

The Byzantine General's Problem was introduced by Lamport, Shostak and Pease.⁶ It is a problem involving N processors in a distributed environment that exchange messages to reach agreement on a value sent by one of them (called the transmitter). In an ideal failure-free system this would pose no problem. However, in an environment where processors may fail, it becomes of fundamental concern in guaranteeing correctness of processing results.

Byzantine agreement protocols have been introduced to solve the problem. A majority of the protocols deal with the execution of consensus algorithms to reach agreement on a transmitted value having a binary domain though there are modifications to the algorithms that allow the domain to be of any size.⁷ When the algorithms are executed, rounds of messages are exchanged over a reliable communication system such that no messages are lost or modified. The sender of any message is always identifiable. All the protocols satisfy the following two conditions:

- (i) If the transmitter is correct and transmits value v then all correct processors must agree on v .
- (ii) All correct processors must agree on the same value.

Variations on the protocols exist in the literature. Some of the algorithms are deterministic³ while others are

probabilistic.² The algorithms can be further divided based on the environment of operation which can be synchronous⁶ or asynchronous.¹ In the completely asynchronous case where there is an unknown bound on message delivery time, processor drift time and message order not guaranteed, it has been shown by Fischer, Lynch and Paterson that no deterministic algorithm exists to tolerate even one failed processor.⁵

Algorithms can also be viewed based on the types of failures they will tolerate. The most severe failure is when faulty processors, also known as traitors, can send spurious messages including forging of information relayed from correct processors. This type of failure is known as "Byzantine Failure." Authenticated Byzantine Failure can occur when messages that are relayed contain unforgeable signatures of the relaying processors. In this case, a message that contains the signature of at least one correct processor can be assumed to be accurate. This limits the damage a faulty processor can accomplish.⁴ Omission failures are failures where the faulty processor fails to relay some of the messages. The simplest failure is known as "fail-stop" where processors simply stop participating in the algorithm.

The number of faulty processors that an algorithm can tolerate and still meet the two conditions is closely associated with the type of failure possible. This paper will focus on algorithms that tolerate "Byzantine Failures." In order for the correctness of the protocol to be insured, it has been shown that the number of traitorous processors must be less than one third of the total number of processors participating in the protocol.⁶

ADA provides a mechanism for inter-task communication through the rendezvous. ADA also provides facilities to handle specific cases of failure of an attempted rendezvous. Some of these include selective accepts, delay and exception handlers to handle tasking error exceptions. For fault tolerant

distributed systems programmed in ADA the loss of an individual processor must not result in system failure.

The purpose of this paper is to present an attempt to use Byzantine protocols to enhance the features provided by ADA intertask communication in the design of fault tolerant systems.

2. DESIGN ISSUES

One approach to implementing Byzantine agreement as a facet of intertask communication assumed the communicating parties to be operating in the same environment. This allowed one to focus on the algorithm and the communication necessary to achieve Byzantine agreement. In this manner a coupling of the agreement protocols and the communication protocols necessary for agreement resulted. The communicating parties were not strongly connected to the agreement protocols that were executing on their behalf and would therefore have little influence on protocol execution except for the initial value transmitted. Since the protocols were executing on behalf of the communicating parties, it was felt that a stronger tie between the two was desirable. As a result a design evolved that had the communicating parties more tightly coupled to the algorithms. The algorithms were also connected to the communication protocols needed to carry them out.

The greatest flexibility was achieved when the algorithm and the communicating parties were strongly connected and the method of communication was supplied by the communicating parties. This allowed for the algorithm to be used to achieve agreement even when the communicating parties were in different environments assuming that the communicating parties handle the inter-environment communication protocols and the Byzantine protocols focus on reaching agreement utilizing the communication protocols supplied.

The rest of the paper focuses on the progression of design issues arising from the loosening of the connection between the agreement and communication protocols and the strengthening of the connection between the agreement protocols and the communicating parties on whose behalf they are being executed.

In order to provide Byzantine agreement as a reusable piece of software the ADA package was chosen as the best means of providing the protocol and all the services needed for utilization of the protocol. Also since flexibility in the number of communicating parties - henceforth known as users - was important a generic package importing the number of

users became the basis of all the designs. The protocols are executed by processes that exchange messages for the purpose of reaching agreement. For this reason the code implementing the algorithm is provided by tasks which have the ability to rendezvous and can thereby exchange messages. The degree of coupling results from the method of task declaration and package instantiation. The tighter the coupling, the greater the affect the user has on algorithm execution and termination.

3. DESIGN PROGRESSION

Coupled Agreement and Communication Package. The first design explored is a package generic on the number of users (N). It contains N identical tasks where each task includes code for the Byzantine agreement algorithm. In addition the package provides the user with the capability of registering and receiving an ID for one of the N tasks responsible for the execution of the protocol on his behalf.

The package allows the user to start the protocol with an initial value which is passed to the Byzantine agreement task that corresponds to the ID supplied by the user. Once agreement is reached the user can retrieve the committed value by calling a procedure provided by the package.

The package will be used as follows: The package is instantiated with a specific value for N (greater than four). Each of the N users register and is returned an ID of a Byzantine agreement task that became active when the package was instantiated. The task is responsible for executing the agreement algorithm on the user's behalf. A user wishing to transmit a value starts the protocol passing the value to the package. Once started, the Byzantine tasks execute the algorithm exchanging rounds of messages between each other until all tasks commit. The user then calls for the committed value. The general design concept is illustrated in figure 1a with package detail illustrated in figure 1b.

It is interesting to note that since all the Byzantine tasks are declared within the package they can rendezvous with each other to accomplish the sending and receiving of messages necessary to reach agreement. To prevent deadlock transport tasks can be used as intermediaries in the process of sending the messages. It is also possible to prevent deadlock through the use of buffers. All the users must be declared within the same environment, the one that instantiates the package.

The first approach has certain

deficiencies arising from the tight coupling of the agreement and communication protocols and the loose connection with the user of the package. The concept of a traitorous user is difficult to understand due to the fact that the user will not influence the content of the messages exchanged once the initial value is transmitted because all the communication occurs within the package between the tasks there. If one of the users should terminate it will not affect the Byzantine agreement tasks since they are declared within the package. This might not be appropriate since the Byzantine tasks are executing the protocol on behalf of the user. In order for the package to be used by users operating in different environments additional support is necessary to provide the inter-environment communication needed.

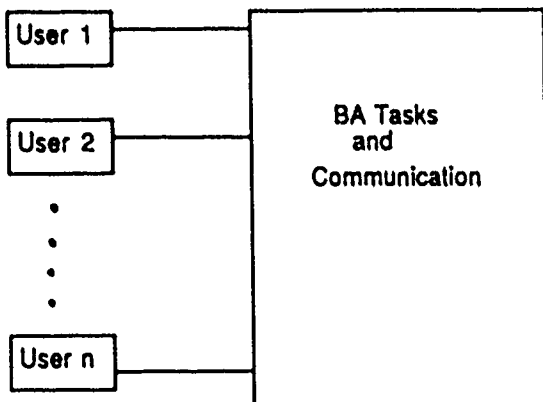


FIG. 1A

package generic on N that offers the user a Byzantine task type that contains the algorithm to be executed to reach agreement. This gives the user the ability to declare objects of this type. It also results in a tighter coupling between the user and the agreement protocol executing on its behalf. The task objects once created must register in order for the package to know how many users are actually utilizing its services. The package will have to provide the tasks some means of sending and receiving messages since they are unaware of each other and therefore cannot rendezvous to exchange messages. Because of this the package has to provide a method of buffering the messages after the send operation and until they can be retrieved. It is also necessary for the package to provide a service enabling the user to start the protocol with an initial value. Once the protocol completes and the tasks commit the user can retrieve the committed value from the task.

Use of the package is described by the following. The package is instantiated with a value for N . Users in the environment instantiating the package can declare up to N Byzantine task objects of the task type provided by the package interface. Once created, the task objects register and wait for the protocol to start. One of the users starts the protocol with an initial value. The task objects exchange messages using the send/receive services provided. These services allow access to the buffered messages within the package. Once the tasks commit the user retrieves the committed value from the task object. Figure 2a illustrates this design concept.

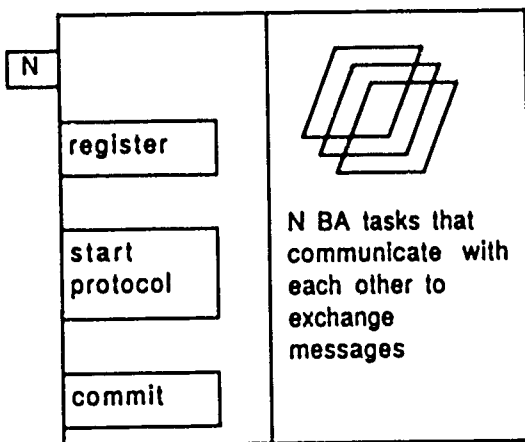


FIG. 1b

Stronger Coupling Between User and Agreement Protocol. In order to improve on some of the conditions resulting from the first design a second approach utilizes a

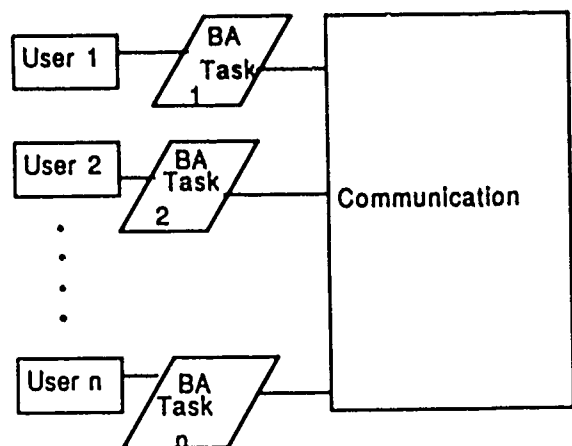


FIG. 2A

Some aspects of the second approach that result from the tighter coupling between the user and the Byzantine tasks

have to be emphasized. Owing to the fact that the user has the ability to declare objects of the Byzantine task type, the task becomes dependent on the user and the user's behavior can influence the task such that if the user is aborted the task will be aborted also. This is not a deficiency since the task is executing on behalf of the user. However, even though there is a tighter coupling between the user and the agreement protocols there is still a strong coupling between the agreement and communication protocols. The send/receive operations are provided by the package and the user still has little influence on the messages transmitted in the various rounds of message exchange. Also by providing the task type in the package interface the user creates tasks that execute the Byzantine agreement protocol with the package providing the communication. This leads to an emphasis on the communication services of the package rather than the agreement protocol. This is seen in figure 2b which illustrates the package details.

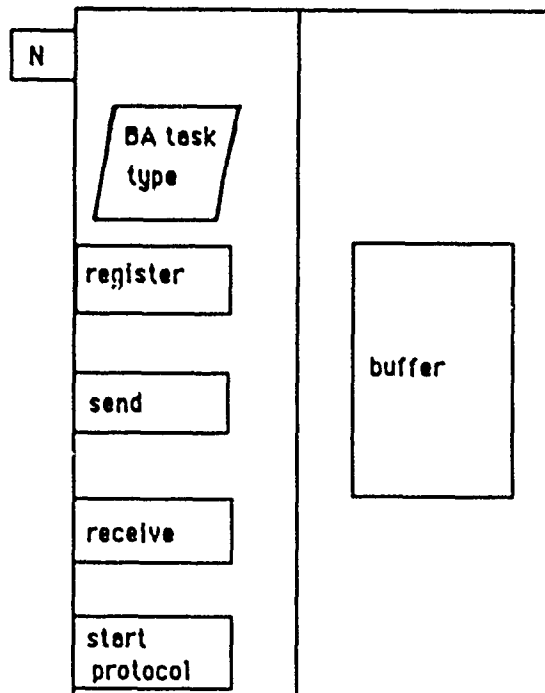


FIG. 2b

Using pointers for Byzantine tasks.

An interesting ramification of the preceding design occurs when the package provides pointers for the Byzantine task. Using pointers allows the Byzantine tasks to be dynamically created. When the protocol starts up, the Byzantine tasks are passed the pointers of all the dynamic tasks participating in the protocol. The pointers can then be used by the tasks to rendezvous for the purpose of sending and receiving messages. Having pointers loosens the connection between the user and

the Byzantine tasks. This results from the fact the allocator is offered through the package interface and objects dynamically created are dependent on the package not the creator of the object.

Imported communication. The last design considered offers a tight coupling between the user and the agreement protocols. This is accomplished utilizing a generic package that imports a discrete range of local IDs to be used by the package to differentiate the various participants, a specific ID within the range to designate the Byzantine task provided by the package and a send operation to be used by the Byzantine agreement task when sending messages. The send operation provided by the user is responsible for possible reformatting of the message supplied by the package to conform to the format expected by the communication protocols as well as possible translation of the local IDs used in the package to IDs used by the communication protocols to designate the various communicators. The send operation must therefore handle messages of a particular message type. Two approaches to the message type are possible. One is to declare the message type as part of the package and the user must supply a send operation capable of handling messages of that type. The other approach is to import a private message type. To allow the Byzantine task the capability of forming messages which is an integral part of the agreement algorithm, the user must also provide a routine to construct the message into the appropriate type using information supplied by the Byzantine task. Once formed the message is then transmitted using the send operation supplied by the user.

The package in addition provides a service that allows the Byzantine agreement task to receive messages from the user and a start protocol procedure that enables the user to start the protocol with a specific value. It also provides a commit routine that the user can call to retrieve the committed value from the Byzantine task.

Package use is illustrated by the following. Each user instantiates the package supplying a range of participating IDs, a specific ID that is within the range to be used to identify the Byzantine task contained in the package and a send procedure. The task in the package becomes active upon instantiation and waits for a user to start the protocol. Once started, the task sends messages using the send procedure supplied by the user and receives messages from the user. It is the responsibility of the user to relay messages from the other participants. Once the task commits the user retrieves the

committed value from the package. This is depicted in figure 3a with the package details highlighted in figure 3b.

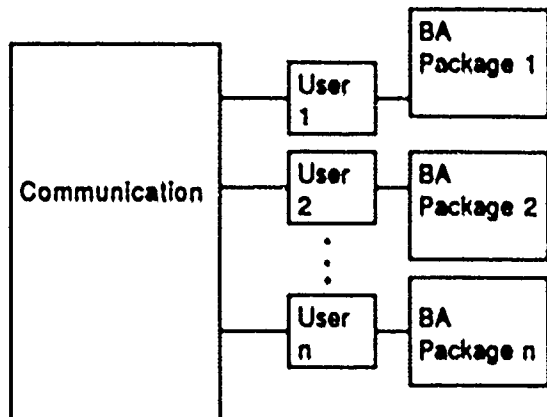


FIG. 3a

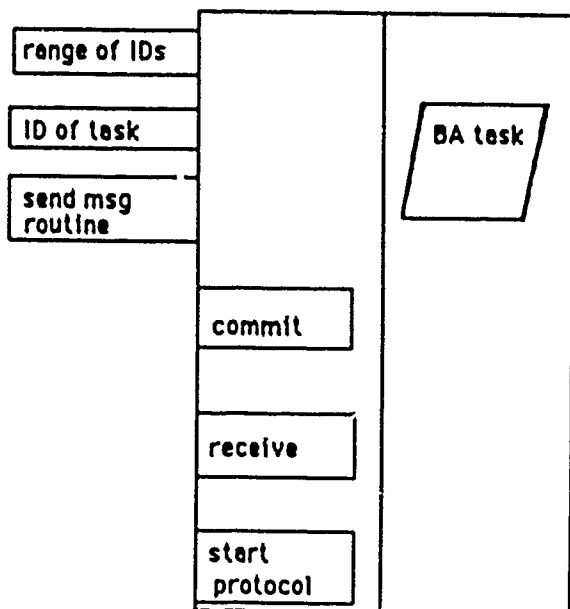


FIG. 3b

This design offers the greatest flexibility in terms of package use. Due to the tight coupling between the user and the agreement protocols fail-stop failures can occur whenever a user is aborted or terminates since the package instantiated by the user is affected also. However, because communication is supplied by the user, Byzantine failures are also possible since the user intercepts all messages sent and received by the package. This gives the user the capability to forge any message. This package can also be utilized by users operating in different environments since it is up to the user to supply the necessary inter-environment communication protocols to the generic package.

The package can be used to provide a synchronous or an asynchronous protocol. This depends on the algorithm the Byzantine agreement task in the package contains. A synchronous algorithm assumes the communication network is synchronous. It is the responsibility of the communication network that the user provides to synchronize message rounds within the package through the user. A package supplying a synchronous algorithm will have to contain a service to advance to the next round that the user can call when necessary to synchronize message rounds based on information from the communication network.

4. CONCLUSIONS

Byzantine agreement protocols provide an important service in distributed processing guaranteeing correctness of processing results. In utilizing them as part of ADA interrask communication one can improve on the reliability of message communication in multi-task processing. By utilizing a package that has a tight coupling between the user and the agreement protocols with the user supplying the communication protocols the greatest flexibility can be achieved in terms of the operating environment and type of algorithm offered.

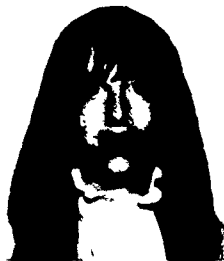
ACKNOWLEDGEMENT The author would like to thank Dr. Jan Bergandy for his helpful comments and editing suggestions concerning the manuscript.

REFERENCES

1. Ben-Or, M. Another advantage of free choice: Completely asynchronous agreement protocols. In Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (Montreal, Quebec, Canada Aug. 17-19) ACM New York, 1983 pp. 27-30.
2. Bracha, G. An $O(\log n)$ expected rounds randomized Byzantine generals protocol. J. ACM (Oct. 1987) pp. 910-920.
3. Dolev, D., Fischer, M., Fowler, R., Lynch, N. and Strong, H.R. An efficient algorithm for Byzantine agreement without authentication. Inf. Control 52,3 (1983), pp. 257-274.
4. Dolev, D. and Strong, H.R. Authenticated algorithm for Byzantine agreement. SIAM J. Comput. 12 (1983) pp. 656-666.
5. Fischer, M.J., Lynch, N.A. and Paterson, M.S. Impossibility of distributed consensus with one faulty process. J. ACM 32 (1985), pp. 374-382.

6. Lamport, L., Shostak, R. and Pease, M. The Byzantine generals problem. ACM Trans. Program. Lang. Syst. 4,3 (July 1982), pp. 382-401.

7. Turpin, R. and Coan, B. Extending binary Byzantine agreement to multivalued Byzantine agreement. Inform. Processing Lett. 18 (Feb. 1984), pp. 73-76.



Shoshana Hartman received her B.Sc. degree in Computer Science from Downstate Medical Center, a division of SUNY, in 1976. She was a programmer for a number of years and has taught Computer Science courses at various schools. She currently has a teaching assistantship at Southeastern Massachusetts University where she is working towards her M.S. degree in Computer Science specializing in software design.

UPGRADING A LISP PROTOTYPE (ADVISE¹) TO A SYSTEM IN ADA¹

Morris Johnson, Kevin Robinson, and Renee Washington

Advisor: Mr. Robert A. Willis, Jr.

Department of Computer Science
Hampton University
Hampton, Virginia 23668

The ADVISOR is an expert system designed to automate and facilitate the duties of a student's academic advisor within a college or university. This system, written in LISP, has the capability to direct students in selecting required courses that are available in their curriculum. It also checks for courses that have prerequisites and corequisites to ensure that they have been taken prior to or will be taken simultaneously with the courses requested. The purpose of this paper is to discuss the problems encountered when converting a LISP prototype to a system written in Ada.

INTRODUCTION

A prototype is usually a quickly built system that performs part or all of the functions desired from a specific software. Converting a prototype to a working system in Ada from any software presents a number of problems. The prototype may be too slow, too large, and/or awkward to use, due primarily to its quick design. Also, some functions that are supported by a prototyping language may not be supported by Ada. When analyzing some of the characteristics of a prototype and the differences between a specialized language and a general-purpose high level language one must confront these problems.

The ADVISOR² prototype is a rule based expert system which accepts a student's identification number and returns a list of courses which the student is eligible to take at that time. The prototype requires the actual code and four rule/database files. The prototype runs within a LISP environment, thus performance is slow due to the interpretive nature of LISP.

We were assigned the task of converting the ADVISOR prototype to Ada in our software design/

development course. During the conversion process, we encountered the following problems: (1) determining how to incorporate the prototype into a software development paradigm, (2) determining when to emulate prototype constructs or when to redesign them for efficiency and (3) to follow good software design practices.

INCORPORATION OF THE PROTOTYPE

Typically software design projects are done in the following manner: Systems Engineering, which entails determining all resources required at the system level; Software Requirements Development, which determines all software functions; and Design, which describes software architecture and procedure to a level of detail suitable to be used for coding, testing, and maintenance. This procedure is commonly referred to as the "Classical Life Cycle Paradigm."³

The prototyping process typically transpires as follows: Definition of the objectives for the software project; Identification of known requirements; Development of an abbreviated design; and Building the prototype. This process is continued until the prototype is completed.⁴ After the prototype has been built, it can be used as the finished product or it can be used to aid in the development of the product in another language.

Our problem entailed the incorporation of the complete prototype within the classical life cycle paradigm. We had to first decide if the prototype could serve as a complete requirements specification. The problems encountered in the process forced us to create a requirements specification based on the prototype. Some of the problems were caused by lack of complete understanding of prototype operation, and a poor knowledge of LISP. Another problem was encountered because the prototype itself existed! The tendency was to underestimate the effort required to develop a complete and accurate requirements specification because it seemed that so much of the work was already done.

DETAILED DESIGN

The basic data structure in LISP is the list. Therefore, a decision whether to emulate LISP constructs or to rewrite them utilizing the features within Ada had to be made. We chose a combination of the two. The list constructs were logically the optimum way to structure some of the run-time data, but the flexibility of other data structures (i.e. trees, queues, etc.) provided more efficient processing. Several problems resulted because of the above combination.

Since the coding was to be accomplished using the Ada programming language, the decision was made to create detailed design documents aligned with Ada concepts. Ada is a strongly typed language which restricts coercion, whereas LISP does not restrict mixing data types. LISP facilitated rapid development of the prototype because little attention had to be given to the actual data types being processed. Using Ada to write the ADVISOR system required specialized packages to create and manipulate binary tree structures, linked-list structures, and queues. The rule bases had to be completely rewritten in order to be utilized efficiently in Ada.

Since the prototype written in LISP used recursion extensively, several decisions had to be made in the design process. We had many problems trying to ascertain when recursion was suitable for use in Ada.

Although the existence of the prototype was beneficial to us during the development of the requirements, it was somewhat detrimental during the design phase. Initially not enough attention was given to the development of the data structures and the internal processing of the system. Consequently, we had to redesign the system several times. Care must be taken not to rely too much on the prototype during the detailed design phase.

Once the detailed design was completed, we began coding the system. The biggest problem we encountered was one of interfacing the

modules. The lists, and many of the variables in the prototype, were global. We had overlooked these factors during the design phase. This oversight resulted in disaster during integration testing, thus necessitating a redesign of the system.

SUMMARY

Utilizing a LISP prototype to develop a software package can cause many problems. This is particularly true when the target language is as strongly typed as Ada. The major difficulties we encountered were the lack of structure and the ease of list processing afforded by the LISP language. The fact that we had a completely working prototype caused us to attempt to emulate too many of its features without full consideration to the eventual Ada implementation. When working from a prototype, designers should be cognizant of these pitfalls. Such errors are more prevalent when using the Ada programming language because of the rigid enforcement of structured program development methodologies.

REFERENCES

1. Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).
2. Copyright 1987 by Willis Computing Services.
3. Roger S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill Book Company, 1987.
4. Robert A. Willis, Jr. Verification and Validation of Artificial Intelligence Systems, TO APPEAR.

An Implementation of the Standard Math Functions in Ada

James Anthony Frush

The University of Mississippi

ABSTRACT

The ADA programming language provides excellent capabilities for writing highly structured, reliable, reusable and easily maintainable applications. However, in order for ADA to be acceptable and usable to engineers and other scientists, certain basic mathematical functions, generally provided as math libraries with other languages, must be made available, reliable, and easily usable. The purpose of this project was to make available a generic package of math functions with enough accuracy and efficiency to encourage the use of ADA by engineers and other scientists at the University of Mississippi. It is also hoped that the package will encourage other students to work with ADA by providing them with challenging material on which to base future projects.

OVERVIEW

The concept of generics in ADA is a powerful feature of the language. This, coupled with other features such as strong typing, ease of maintenance, and support for parallel processing make ADA superior to FORTRAN for engineering and scientific applications. The project is implemented as a generic package of math functions so that the programmer may instantiate the package for any floating-point type. The package contains functions for the computation of the following: Square root, Cube root, Sine, Cosine, Tangent, Cotangent, Natural Logarithm, Base Ten Logarithm, Power, Exponential, Arcsine, Arccosine, Arctangent, Hyperbolic Sine, Hyperbolic Cosine, and Hyperbolic Tangent.

In deciding how to implement this package, several factors had to be considered. First, The time allotted for completion of the project was limited to one semester.

This time constraint effectively ruled out starting from scratch. Secondly, there were two compilers available for the University of Mississippi's mainframe.

The first compiler, the Telesoft Telegen2 version 1.0/3.05 limits the programmer to six decimal digits of accuracy for floating-point types. The second, the Alsys IBM 370 ADA compiler for VM/CMS, version 2.3, allows up to eighteen decimal digits of accuracy for floating-point types. Since the primary users of this package are engineers, it was necessary to provide the maximum accuracy possible, therefore, the Alsys compiler was chosen to develop and test the package.

In order to satisfy the time constraint, a search for any existing packages which might satisfy the project's requirements was begun. Fortunately, there was a package of math routines available on a tape containing files from the Ada Software Repository which met the project's requirements precisely.

The package was written in 1982 by LCOL William Whitaker, a member of the original High-Order Language Working Group, and revised in 1986 by LT Tim Eicholz. The package is an implementation of the functions found in William Cody and William Waite's "Software Manual for the Elementary Functions", Prentice Hall, 1980.

The tapes containing the files from the Ada Software Repository were in DEC-VAX format and had to be translated to IBM format to be usable. Using a program originally intended to perform the translation from the PDP-11 to IBM format, the files containing the math package and test routines were translated and it was discovered that many lines of code were missing. These lines were painstakingly reconstructed from the Cody-Waite manual.

PRECISION

The algorithms in the Cody-Waite manual provide polynomial approximations that are accurate to about eighteen decimal digits. Since the compilers available limit the definition of

a floating-point type to eighteen decimal digits, the Cody-Waite approximations provide the maximum accuracy available at this time. As compilers allowing more than eighteen decimal digits become available to the University, The functions may be easily modified to provide the higher accuracy.

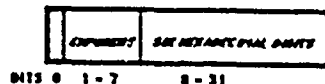
In order to understand the precision considerations for the package, a brief review of some fundamentals of floating-point representations in general, and the representations used by the IBM 370 in particular, would be helpful.

Please keep in mind that a floating-point number can be defined as follows:

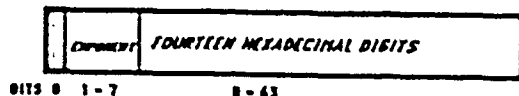
SIGN * MANTISSA * RADIX ** EXPONENT

In this representation, SIGN is either -1 or +1, MANTISSA is a normalized fraction, (that is the first digit following the decimal point is not zero), RADIX is the base of the representation, (i.e. 10 for decimal, 2 for binary, 16 for hexadecimal, etc.), and EXPONENT is a positive or negative integer. The IBM 370 architecture represents floating-point numbers as hexadecimal digits. In other words, the radix in the above representation is 16 on the IBM 370. Two other considerations must be kept in mind when dealing with floating-point numbers on the IBM 370. First, the exponent is expressed in excess 64 notation, (the seven bit exponent is treated as a binary value and the exponent is derived by subtracting 64 from it). Second, IBM provides three formats for representing floating-point numbers, these are:

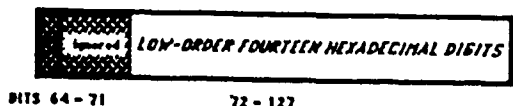
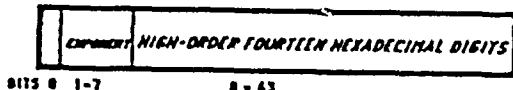
SHORT FORMAT



LONG FORMAT



EXTENDED FORMAT



The number of decimal digits available for a given number of hexadecimal digits can be found by solving for X in the following equality:

$$10^X = 16^n$$

$$X = n \log_{10} 16$$

Using this equation it is easy to see that for the IBM short format, 7 decimal digits are available, for the long format 16 decimal digits are available, and for the extended format 33 digits are available. The Alsys ADA compiler maps ADA floating-point types onto the IBM formats in the following manner: the predefined floating-point types SHORT_FLOAT, FLOAT, and LONG_FLOAT, are represented in the IBM short, long, and extended formats respectively, however, for decimal accuracy these three types are limited to 6, 14, and 18 decimal digits respectively. It is assumed that this was done to prevent loss of accuracy due to rounding error. As for the user-defined floating-point types in ADA, they are represented in the appropriate IBM format. For instance:

type MY_REAL is digits 15;
type YOUR_REAL is digits 12;

In this example, MY_REAL would be represented in the IBM extended format and YOUR_REAL in the long format. Even though a number with 15 decimal digits could be represented in the IBM long format, the results of computations on this number could contain errors in the least significant digits. By representing the number in the extended format, errors occurring in the least significant digits will not degrade the accuracy of the defined type. It should be made clear that the predefined types SHORT_FLOAT, and LONG_FLOAT are not required by the language definition but are provided by the Alsys environment in keeping with recommendations in the language reference manual.

The algorithms in the Cody-Waite manual recommend subtle variances to be used on machines using different floating-point representations. (i.e. binary, octal, hexadecimal) While these variances do cause minor differences in accuracy, by using the algorithms for binary machines and extracting a base two representation of the floating-point numbers, the package used in my project provides portability and results that are accurate within the defined type.

Now that we have some of the preliminaries out of the way, let's take a look at a representative example of the functions available in the math package. For the purposes of this discussion the natural logarithm function will be used as an example of the other functions in the package.

Since the focus of this paper is not numerical methods the algorithm will be discussed in very general form and a copy of the code will be provided for those who wish a more detailed consideration.

In order to calculate the logarithm, three steps must be taken:

1. Reduce the argument to a small logarithmically symmetrical interval around 1.
2. Compute the logarithm for the reduced argument using a polynomial approximation (Cody-Waite use a minimax rational approximation generated especially for their work).
3. Reconstruct the desired logarithm from its components.

Attempts to calculate the logarithm for invalid arguments are handled in various ways. An attempt to calculate the logarithm for a negative value causes the function to send an error message to the standard output and calculate the logarithm of the absolute value of the argument. Similarly, an attempt to calculate the logarithm of zero causes the function to send an error message to the standard output and return the largest negative number representable by the machine. Any other exception raised in the logarithm function causes an error message to be sent to the standard output and a value of zero to be returned. The above technique, employed wherever possible throughout the package, prevents a complete crash of the program while notifying the programmer that an error has occurred.

TESTING

After reconstructing the missing lines of code from the package it was compiled in its generic form. However, any attempt to compile a program that instantiated the package caused the compiler to crash. The problem turned out to be a documented bug in that version of the compiler. While waiting for a new version of the compiler to be delivered, the package was compiled and tested as a non-generic for all floating-point types. Two types of testing for accuracy were performed. First, the Cody-Waite test routines

provided with the package were used. The test routines use 2000 random arguments along with various arguments used to exercise the exception handlers. The tests for accuracy are performed using various mathematical identities.

Returning to the natural logarithm function, for example, one accuracy test performed measures the maximum relative error in the identity:

$$\ln(x) = \ln(17x/16) - \ln(17/16)$$

so we measure the error E as

$$E = (\ln(x) - (\ln(17x/16) - \ln(17/16))) / \ln(x)$$

After running these test routines it was found that at no point did the maximum relative error exceed the digits of accuracy specified in the floating-point type definition.

The second type of accuracy testing involved comparing the results to accepted values. Although not yet complete, the comparisons performed so far show that this package provides exactly the same results as the IBM VS/FORTRAN version 3.0 routines currently in use at the University of Mississippi. A comparison of the results obtained from the package for several different floating-point types against tables of values produced in 1941 for the city of New York under the sponsorship of the National Bureau of Standards indicates that the values are completely accurate.

Testing the efficiency of the functions has not, as of this writing, been initiated.

SUGGESTIONS FOR IMPROVEMENT

As stated in the abstract, one of the goals of this project was to encourage students to pursue future projects in ADA by providing them with significant material to start with. Using the natural logarithm once again as a representative example, some suggestions for improvement of the package should be discussed.

The logarithm function uses two separate approximations, one for floating-point types of less than ten digits of accuracy and one for those with more. This clearly does not make full use of ADA generics. Modification of the functions to provide separate approximations for a wider range of floating-point types would not be difficult. Modification of the functions to provide more than eighteen decimal digits of accuracy, (whenever a compiler capable of this becomes

available to the University), would be another significant improvement. Since all of the functions in the package are separately compiled, modification of the package could be performed by individuals or teams easily and efficiently.

CONCLUSION

This package, as it stands, provides a reliable and accurate base of mathematical functions. It is hoped that the availability of this package will encourage engineers and other scientists at the University of Mississippi to take advantage of some of the features of the ADA programming language. The package has been ported to the IBM PC/AT and, as other compilers become available, will be ported to the various machines at the University, including the Cyber supercomputer. Finally, It is my strong desire that students will find this fertile ground for challenging and significant projects.

REFERENCES

Amaroso, Serafino and Inargiolo, Giorgio, ADA: An Introduction to Program Design and Coding, Pitman Publishing Co., Boston, MA, 1985.

Cody, William J. and Waite, William, Software Manual for the Elementary Functions. Prentice Hall, Inc., Englewood Cliffs, NJ, 1980.

Ford, B. et.al., Scientific ADA. Cambridge University Press, London, 1986.

Kudlick, Michael B., Assembly Language Programming for the IBM System 360 and 370. William C. Brown Co., Dubuque, IA, 1983.

Luke, Yudell L. The Special Functions and Their Approximations, VI - XII. Academic Press, New York, NY, 1969.

Rehmer, Karl, "Development and Implementation of the Magnavox Generic ADA Basic Mathematics Package", Ada Letters, v.7, no.3, May-June 1987.



James A. Frush is a Senior Majoring in Computer Science at the University of Mississippi's School of Engineering. Mr. Frush is due to receive a Bachelor of Science degree in August 1989.

Decomposition Schemes for Static and Dynamic Analyses of Ada Programs

Rajeev Gopal
Department of Computer Science
Vanderbilt University
Box 5632, Station B
Nashville, TN

Abstract

Decomposition schemes like stepwise refinement, and modular and structured programming have been effectively used in the program design and implementation phases of software development. In this paper, the role of program decomposition formalisms for analyzing the static and dynamic properties of large programs is identified. These decompositions can assist in program comprehension, testing, debugging and verification. We provide the VFG (Visibility and Control Flow Graph) based intermediate representations for Ada programs, at both intra-module and inter-module levels. An operational approach for describing the visibility and scoping rules of Ada, with respect to the visibility flow graph representation of Ada programs, is also presented. We describe how this can be utilized to elicit information about the static semantics of a program. Dependence relations, for both static and dynamic analyses, are used to define program projections which may be smaller than the program itself. These projections can be generated in an on-demand fashion and executed displaying the dynamic semantics of the original program. The usefulness of these decomposition schemes for program debugging and verification is also discussed.

1 Introduction

Evolution is an intrinsic behavior of all natural and artificial systems. Software systems also evolve in order to maintain compatibility with the domain in which they are embedded. Automated support for software evolution is mandatory for large systems in order to provide cost effective computerization in the domain of application. For the design and implementation phases of software development, techniques like stepwise refinement, modular decomposition and structured programming are being successfully used. The research in the various aspects of software engineering has been effectively utilized in prototyping programming environments that are rapidly being introduced for software production.

These novel programming paradigms, techniques and tools may lead to significant productivity gains for developing new systems. Their role in maintaining existing software that is evolving, or software being developed in the traditional way, is rather limited. The cost of software maintenance now amounts to more than 50% [1] of the overall software costs of about 100 billion dollars per year [4]. In the absence of a complete and consistent documentation of the requirements and design specification of the software system, often the source code is the only genuine representation of the system and it has to be analyzed and understood for any maintenance activity [3].

We are studying the nature and effectiveness of program decomposition techniques in the context of software evolution (or software maintenance). This research is used towards the development of a program analyzer that can automate the techniques for decomposing large multi-module programs and simplify their static and dynamic analyses. Although the analyzer provides a new perspective to the program comprehension and debugging processes, we envision its application in a wider domain of automated support for program development, testing, and verification efforts.

The paper can be broadly divided into two parts. In section 2, we begin with a discussion on the visibility and scoping rules of programming languages. This sets the stage for presenting our approach towards managing and displaying static information about programs. In section 3, we provide the Visibility Flow Graph (VFG) model as an intermediate representation for large multi-module programs. This model can be used to elicit information about the static properties of programs. This is covered in section 4.

In section 5, we consider the Control Flow Graphs (CFG) that provide the basis for the decomposition schemes for dynamic properties. In the next section, we introduce program projections in both static and dynamic contexts. We use dependence relations among the statements and the variables of a program for defining and generating these projections. Static projections are useful for program verification and their dynamic counterparts are suited for program testing and debugging with actual input values for the variables.

We present customized VFGs and CFGs for Ada programs in section 7. This is followed by some of the salient features of a prototype program analyzer for Ada programs. In section 9, we briefly discuss the potential of our approach for supporting software evolution. Finally, we address the current status of this project and provide some directions for future work.

2 Visibility Control in Programming Languages

In programming languages, the visibility issues are described in terms of declaration of entities and the related scope (the portion of the program text where a declaration is potentially visible) of these declarations. In a declaration, a software entity is associated with a name. Most languages allow the usage of the same name in multiple declarations and the scopes of various declarations can overlap. Some languages provide overloading of names as well. Thus, any description of visibility control should include the declaration and naming mechanisms with visibility and overloading rules.

A program may use many of the visibility constructs provided by the language. For our discussion, we shall consider the Ada programming language. The concept of a declarative context is important in this approach. A program is translated into a graph, each node of which corresponds to a declarative context, interconnected with arcs representing the visibility flow. The notion of declarative context can be traced back to early languages like FORTRAN, that support the declaration of subprograms with locally declared variables which are not visible outside the subprogram. In ALGOL 60, a declarative context is formally defined, along with other aspects of the language. Some obvious examples of declarative contexts are *function* and *struct* in C, and *procedure*, *function* and *record* in Pascal. Among more recent languages, Ada includes *package* and Modula has *module* as new types of declarative contexts. Object-oriented languages like SmallTalk have *class* and *object* primitives that correspond to declarative contexts.

A declarative context provides a framework for declaring new entities and doing computation (for procedural entities) using these entities. The declaration of an entity may require a user defined type that may or may not have been locally declared. Similarly, an entity used in the computation part can also be non-local. For this reason, programming languages provide visibility flow mechanisms to import non-local entities from other declarative contexts. These mechanisms can be implicitly present, as in the case of procedure nesting, or an explicit language construct can be used like the *with* clause to import the public entities of an Ada package.

2.1 Implicit Nesting

In ALGOL 60, nesting is the primary visibility mechanism for facilitating the use of non-local entities across the declarative context boundaries. The notion of nesting and the associated flow of visibility from the parent context to a child context (and not vice versa) is inherited by several other block structured languages including Pascal, Modula-2 and Ada.

2.2 Linear Elaboration

In many languages like Pascal, Ada and C, linear elaboration is another visibility mechanism that implicitly controls the visibility of entities within a declarative context. With linear elaboration, it is illegal to use an entity that has not been declared, at least partially, before the entity is actually used. Modula does not depend on linear elaboration and the visibility is uniform across the entire declarative context.

2.3 Explicit Nesting

Traditionally, nesting is implied when a child declarative context is declared within the parent declarative context. In Ada, nesting can be explicitly specified with the *is separate* and *separate* clauses. They support separate compilation of packages and procedures, while maintaining visibility due to implicit nesting.

2.4 Structured Types

An entity declared to be of structured type gets all the fields (or the subcomponents) of the associated type. Thus *struct* type of C and *record* type of Pascal and Ada, both have inherent source of visibility that can be imported by other entities. In constructs like *module* of Modula-2, it is possible to associate procedural code as a subcomponent of a structured entity. These language primitives can be used for direct implementation of *abstract data types*. Further, languages like SmallTalk extend this abstract data type implementation mechanism with *class* constructs. Here it is possible to inherit subcomponents from other contexts using *object inheritance* rules that include transitivity. These object-oriented languages can support an entire hierarchy of contexts related through visibility flow relations [6].

2.5 Previous Work

Traditionally, visibility control mechanisms have been described informally. It is a common practice to provide BNF (context free) descriptions for representing the syntactical features of programming languages [1]. Formal methods do exist, however, for representing the non syntactical aspects (that includes visibility control) of programming languages [10]. In [13] the disadvantages of this formal approach for describing the visibility rules are mentioned and a visibility graph

model has been proposed. Visibility mechanisms are described using the concepts of *requisition of access* and *provision of access*. This model is aimed at language designers for justifying new mechanisms and programmers for identifying the suitability of a mechanism within a program. It also provides a graphical representation of visibility properties of a program.

3 Visibility Flow Graphs

The Visibility Flow Graphs consist of structured nodes and edges interconnecting the nodes. A node corresponds to a declarative context (DC) while the edges interlinking DCs depict visibility flow information. Within a declarative context, a sequential list provides the skeletal structure connecting various units, of type declarative or requisitionary. A unit may correspond to an atomic program construct like a variable declaration (declarative) or a statement (requisitionary). A non-atomic unit is associated with another declarative context that may participate in the visibility flow information of the current declarative context. We define the visibility, declaration, requirement and provision sets for the entities within the framework of VFGs.

The VFGs display the interconnection of the declarative contexts in a program. Depending on the programming language and the associated visibility constructs used, a declarative context may correspond to an entire program, a module, or a part of the module. This representation is uniform across the inter and intra-module visibility representation. This uniformity is in sharp contrast with the duality of "large" and "small" programs, associated with the Module Interconnection Language (MIL) approach [9] towards large programs.

Atomic units never exist on their own and are always contained within a DC. An entity may have several attributes but usually its name is sufficient for this discussion of the VFG model.

3.1 Declarative Contexts and Units

The declarative contexts constitute the nodes of the visibility graphs. A context is structured as a sequential list¹ of units corresponding to declarations and statements. A declarative unit may declare a new entity (e.g., a variable) and may require other entities (the type). A unit corresponding to a statement may require access to other entities (like variables, procedures etc.) Thus, there are two attributes of each unit. The set D_i includes the entities declared in unit i . Similarly, R_i corresponds to the set of entities that are required by unit i .

To accommodate linear elaboration, each unit is assigned an index number, denoting its position in the sequential unit-list associated with a declarative context. This ordering can readily be determined for the declarative part of a context as any non-atomic declaration is treated as a different declarative context. For the procedural part, it is necessary to provide an ordering function to accommodate compound statements like conditionals and loops [6].

3.2 Visibility Function for a Context

Using the definitions of units and the ordering function, the visibility function V can be defined at the end of unit i of a declarative context.

$$V_i = (V_{i-1} \cup V_i^* - D_i) \cup D_i \quad (1)$$

¹All programming languages employ this sequential juxtaposition of declarations and statements. Although for statements, this sequential nature provides the control flow, its importance for visibility purposes is relevant only when linear elaboration is effective. Thus, for Modula-2 declarations the sequential aspect of this list is meaningless.

Here, V_{i-1} corresponds to the visibility value after unit $i-1$, and D_i corresponds to the entities provided by the i th unit. The term V_i^* denotes the visibility imported from other contexts and should be known before V_i can be effectively determined. Subtracting D_i takes care of visibility hiding due to a redeclaration of an entity at unit i that was already visible from some non-local context. For languages not using linear elaboration, $V_i^* = V_i$ for all i .

Thus V_i corresponds to both, the visibility before unit $i+1$ and just after the i th unit. V_0 corresponds to visibility before the first unit and includes only V_0^* that is imported from other declarative contexts.

3.3 Importing and Exporting Visibility

In the definition of the visibility function V within a declarative context, the term V^* corresponds to the visibility imported from other contexts. To accommodate linear elaboration, V_i^* is defined for each unit. In a VFG, an arc (M, N) corresponds to the flow of visibility from the declarative context M (the source) to the context N (the destination). The context N is said to import visibility of entities exported by the context M . With linear elaboration the interpretation of visibility import is dependent on the actual position of the flow arc within the destination context. Thus, the target of the arc (M, N) is qualified with index i . The following equation describes the imported visibility for a declarative context N :

$$V_i^* = \bigcup_{(K, N) \in VFG_{i-1}} IN_{(K, N)} OUT_{(K, N)} \quad (2)$$

Here, IN is a function defined at the targets of the VFG edges. Similarly, OUT , defined at the source nodes, describes the export of entities from the declarative context M to N . The source M is qualified with index j . With the definitions of the functions IN and OUT , that are language dependent, it is possible to translate the idiosyncrasies of various language-specific rules that govern visibility flow across the declarative contexts.

3.4 Requirement and Provision Sets

The set R_i contains all the entities that are required in unit i . The union of these sets over all units in a context leads to the requirement set corresponding to that context, it is defined as: $R_N = \bigcup_{i=1}^n R_i$. Similarly, $D_N = \bigcup_{i=1}^n D_i$, that describes the set of all local declarations. If the requirements of a context can be met locally then $R_N \subseteq D_N$ and for linear elaboration: $\bigcup_{i=1}^n R_i \subseteq \bigcup_{i=1}^n D_i$ for $k = 1..n$.

For a well formed program, it is essential that all the required entities, within a declarative context, should be provided locally or imported. This is equivalent to saying that the relation $R_i \subseteq V_i$ be true for all i .

The provision set, corresponding to an entity e , is defined as follows:

$$P(e) = \{f | e \in R_f\} \quad (3)$$

The entities e and f could be declared in the same or different declarative contexts. The information contained in provision sets is a generalization of the traditional cross-referencing outputs.

4 Generating Secondary Information from VFGs

A VFG stores the raw information about the corresponding program. It can be used to generate secondary information, for answering on-demand queries. In this section, we describe this generation process. This can provide the core for implementing the primitives of a program query language [6] to elicit information about static semantics of programs.

4.1 Local Visibility in a Declarative Context

All the local declarations, subject to linear elaboration, provide local visibility for a declarative context. The generation of local visibility requires a single pass over the unit-list of the declarative context. This can readily be determined using equation 1 iteratively for each i .

4.2 Global Visibility in a Declarative Context

Global visibility in a declarative context can be determined using equation 1. Before the visibility function can be generated for a specific DC, it should be known for all the declarative contexts that export visibility to the DC under consideration. The VFG edges (describing visibility flow) impose a partial order on the VFG nodes and it is possible to predetermine all such DCs from which the current DC is importing visibility. This scheme ensures that only the directly affected part of the VFG of a program has to be regenerated when the program is incrementally modified.

4.3 Requirements in a Declarative Context

The determination of the requirements of a declarative context is the union of requirement sets of all the units. For a well formed program, the requirements should be provided by the visible entities, as described by the global visibility function. If the program is not complete, then it is meaningful to define an *IMPORT* set, for the program. The aggregation of all unresolved requirements in a program constitutes the *IMPORT* set. It can be generated by collecting all $R^d = V^d$, where R^d is the requirement set of DC d . Thus, $IMPORT = \bigcup_{d=1}^n (R^d - V^d)$. Here " $-$ " is the set subtraction operator. This import set for a module is the *REQUIREMENT* set defined and used in the MLL approach [9].

4.4 Provision Sets

The provision set for an entity can be determined using equation 3. For this, each requirement of all declarative contexts that can import visibility from the current DC, must be considered for resolution with the local or imported visibility. The aggregation of provision sets of all declarative contexts constituting a module is a more generalized form of the *PROVISION* set defined and used in the MLL context [8], and the traditional cross-referencing of programs. A provision set contains the entities that are being provided to other contexts (or parts of a program). It includes the actual entities that have unresolved requirements, and the declarative contexts containing them. It should be distinguished from the set of entities that are visible in other contexts (or modules). That information directly corresponds to visibility or potential provision of entities for resolving some requirement.

5 Control Flow Representation of Programs

Control flow graphs have been used in various contexts related to compilation and optimization of programs [2]. The control flow properties may not be explicitly present in the syntax trees corresponding to a program. During the compilation of a program, the syntax tree is only an intermediate representation for the program syntax. The syntax tree is augmented with the control flow information that is explicitly stored.

The CFG is defined recursively. At the highest level, it is a sequential list of statements of types simple and compound. A node could be atomic if it corresponds to a simple statement like assignment. For compound statements like conditionals, it corresponds to a control flow subgraph that includes the condition, the then part and the else part.

The CFG is a tree for programs without any arbitrary goto statements. Each subunit of such a program has a single entry and exit points. Each assignment, procedure call and null statement is a simple node, and loops and conditionals correspond to a subtree.

The CFG of a program provides the framework for program analysis and has attributes for defining and determining additional properties pertaining to static and dynamic analysis of the program. As discussed in the next section, the attributes include the dependence relations V_P , S_P and V'_P that are used for generating static and dynamic projections. In addition, there are attributes for monitoring the program trajectory while it is executing. These in conjunction with the dependence relations are used in generating more precise program projections.

6 Program Projections

In this section, we define program projections of two types: static and dynamic. Dependence relations are used for this purpose. We define three such relations and describe how they can be determined for a program in both static and dynamic contexts. These relations can also be used in proof maintenance and testing maintenance.

We define program projections to be a subset of all program statements, maintaining the original order and a part of the overall semantics. A projection P_P of program P is defined for a variable v , $v \in V$, and from statement S_1 to S_2 . The aim is to construct P_P with only those statements from the sequence S_1 to S_2 such that the value of v is same after executing P_P as it would be after statement S_2 when P is executed.

Static analysis has been used earlier to generate useful information about programs for compilation, optimization, testing, and understanding purposes [3]. Projections can be defined based only on the static analysis of the program, similar to program slicing [12] or partial statements [3]. Such projections, however, have limited applicability while executing a program with actual values for testing and debugging purposes. In this section, we define the dependence relations that can be used in the context of both static and dynamic projections of programs. Projections defined in dynamic contexts are more precise compared to their static counterparts.

6.1 Dependence Relations

We define three relations that are useful in capturing the static and dynamic properties of a program.

- $S_P : (s, v) \in S_P$, if statement s depends on (uses) the input value of variable v .
- $V'_P : (v, s) \in V'_P$, if the output value of variable v depends on execution of the statement s .
- $V_P : (v, u) \in V_P$, if the output value of v depends on the input value of u .

While considering the dependence relations of multiple programs (or multiple parts of the same program), a superscript is used to associate a program with its relations. Thus, V'_P denotes the V'_P relation of program P .

Program projections can be generated using the second dependence relation, V'_P . For a program P and variable v :

$$P_P = \{s | (v, s) \in V'_P\} \quad (4)$$

For these relations, the input and output values correspond to the state before and after executing the program (or its parts). These dependence relations can easily be defined for simple statements. Other statements can be visualized as a sequence of smaller statements. The following definitions for the modification and preservation of variable v in a program P will be used later for generating these relations:

MOD = $\{v | v$ is modified by the program $P\}$

PRE = $\{v | v$ is preserved (not modified) in $P\}$

Relation V_P for a program P can be defined in terms of S_P and V'_P as follows. For variables v and u , $(v, u) \in V_P$ if any of the two following conditions is satisfied:

1. The variable v depends on s (or, $(v, s) \in V'_P$) and the statement s in turn depends on variable u , i.e., $(s, u) \in S_P$.
2. The variable v is preserved by the program P .

This leads to:

$$V_P = V'_P \cup P_P \quad \text{where } P_P = \{(v, u) | u \in \text{PRE}\} \quad (5)$$

Here, $v \in V$, the set of all variables in program P and the composition operator " \cup " has a higher precedence than \cup and \cap .

6.1.1 Null statement

A program P that consists of a single null statement does not modify any variable. For all $v \in V$, the output value of v depends only on the input value of v . Thus, $(v, v) \in V_P$ if and only if $v = u$. This results in $V_P = I$.

Other relations are obvious as there is no assignment in a null statement and all the variables of the program P are preserved. Thus, MOD = \emptyset , PRE = V , $S_P = \emptyset$, and $V'_P = \emptyset$.

6.2 Assignment Statement

For an assignment statement,

$s: v := \text{expr};$

v is the only variable that is modified and it depends on all the variables appearing in the expr part of the statement. Therefore, PRE = $V - \{v\}$, MOD = $\{v\}$, and relation $S_P = \{(s, u) | u \in \text{expr}\}$. The output value of variable v depends on the execution of statement s so, $V'_P = \{(v, s)\}$. The relation V_P can be obtained by using equation 5

$$V_P = \{(v, u) | u \in \text{expr}\} \cup \{(v, u) | u \in V - \{v\}\},$$

that is equivalent to:

$$V_P = \{(v, u) | u \in \text{expr}\} \cup \{v - (v, v)\}.$$

6.2.1 Sequence of Statements

Earlier the relations have been defined for programs containing single assignment and null statements. Definitions for loop and conditional statements are provided later. Procedure and function calls can be accommodated by considering them as generalized assignment statements. These basic definitions can be used to construct relations for programs containing multiple statements where the program is visualized as a finite sequence of statements. Since these relations satisfy the associativity rule, it is sufficient to show how to generate relations for a sequence of two programs. Using induction, a program of arbitrary size can be analyzed and relations can be constructed.

Consider the sequence P of two programs P_1 and P_2 , where all the three relations, V_1^P , S_1^P and V_2^P are known for each of P_1 and P_2 . For the sequence, $PRE^P = PRE^{P_1} \cap PRE^{P_2}$ and $MOD^P = MOD^{P_1} \cup MOD^{P_2}$. It is assumed that both P_1 and P_2 have single entry and exit points.

Now $(s, v) \in S_1^P$ if any of the following two conditions hold:

1. The statement s is in P_1 and its execution uses (depends on) the input value of variable v ; i.e., $(s, v) \in S_1^{P_1}$.
2. The statement s is in P_2 and it depends on the input value of variable u and the output value of variable w , after P_1 , depends on the input value of variable v before P_1 . This is equivalent to saying that $(s, u) \in S_1^{P_2}$ and $(u, v) \in V_1^{P_1}$.

Thus,

$$S_1^P = S_1^{P_1} \cup S_1^{P_2} \cdot V_1^{P_1} \quad (6)$$

Similarly, the relation V_2^P can be obtained for the program P .

$$V_2^P = V_2^{P_1} \cup V_2^{P_2} \cdot V_2^{P_1} \quad (7)$$

Finally, equation 5 can be used to determine V_1^P .

$$V_1^P = V_2^P \cdot S_1^P \cup (P_1^{P_1} \cap P_2^{P_1}) = \dots = V_1^{P_1} \cdot V_1^{P_1} \quad (8)$$

6.2.2 Compound Statements

The dependence relations can also be determined for compound statements. However, unlike assignment and null statements, the definitions are different for static and dynamic contexts. For example, unless the program is executed for specific initial values for the input variables, it is not possible to determine whether the *then* part of a conditional will be executed instead of the *else* part.

Conditionals For a program P , having a single *if* statement,

s : if *expr* then P_1 else P_2 end;
 $MOD^P = MOD^{P_1}$, $PRE^P = PRE^{P_1}$,
 $V_2^P = MOD^{P_1} \odot s \cup V_2^{P_1}$,
 $S_1^P = \{s\} \odot COND \cup S_1^{P_1}$, and
 $V_1^P = V_1^{P_1} \cup (MOD^{P_1} \odot COND) \{0\}$.

Here $COND$ denotes the set of variables in *expr*, " \odot " is the cartesian product operator, and i could either be 1 or 2 depending on whether *expr* evaluates to TRUE or FALSE.

Loops Suppose the program P contains a single loop statement,

s : while $COND$ loop p end;
 If the loop iterates for $N+1$ times then:
 $MOD^P = MOD^P$, and $PRE^P = PRE^P$.

The dependence relations are as follows:

$$S_1^P = (\{s\} \odot COND \cup S_1^P) \cdot V_1^P \odot N,$$

$$V_1^P = (MOD^P \odot COND \cup I) \cdot V_1^P \odot N, \text{ and}$$

$$V_2^P = MOD^P \odot \{s\} \cup (MOD^P \odot COND \cup I) \cdot V_1^P \odot N \cdot V_2^P.$$

Here $V_1^P \odot N = \underbrace{V_1^P \cdot V_1^P \cdot \dots \cdot V_1^P}_N$

6.3 Static Relations

In general, it is not possible to predict the actual path of computation with static analysis of a program. All control flow paths are considered while generating these dependence relations for static projections. With static analysis only, for $(v, s) \in V_2^P$ the final value of variable v after executing program P may depend on the execution of statement s . This relational approach for static analysis was introduced in [3] and the definition of our dependence relations is motivated by the λ , μ and ρ relations of [3] and their usefulness in generating information about live variable analysis, redundant code elimination, expression movement and generation of partial statements (conceptually similar to static program projections). These relations can be generated by parsing the program text and identifying all variables, statements, structure of all compound statements and the sequencing of statements.

6.4 Dynamic Relations

The dependence relations that were defined for static analysis are also useful during the dynamic execution of a program with actual values. Dynamic projections can be more precise as the actual trajectory of the program execution is also available to the analyzer.

We describe a procedure that generates the dynamic dependence relations for the program P as it executes, under the control of the analyzer. This procedure is called by the analyzer after executing each statement of P . An internal stack is maintained by the procedure for processing compound statements. The generator maintains the dependence relations for the part of the program that has already been executed. This information is preserved between any two invocations of this procedure.

The stack can store the following items:

1. statement type
2. expression *expr* for *if* and loop statements
3. dependence relations v_i , v_r and s_r
4. sets *Mod* and *Pre*

Pre-Processing

Initialize the dependence relations

$v_i = i$;
 $v_r = \phi$;
 $s_r = \phi$;
 $Mod = \phi$;
 $Pre = V$;

procedure DynamicDependenceGenerator(*s* in statement)
 begin

```

case x is
  when assignment-statement =>
    update  $v_i, s_i, v_i$ ;
    update Mod and Pre;

  when begin-of-if-or-loop-statement =>
    push statement-type;
    push expr part of if or loop statement;
    push  $v_i, s_i, v_i$ ;
    push Mod and Pre;
    initialise new  $v_i, s_i$  and  $v_i$ ;
    initialise new Mod and Pre;

  when end-of-if-or-loop-statement =>
    complete the relations from the current
    values and top-most entry (for expr, and type);
    unstack the top-most entry and make it current;
    update current relations with the recently
    completed relations for a compound statement;

end case;
end;

```

6.5 Example

Consider the program that generates fibonacci numbers for which the dependence relations and projections are provided below.

```

procedure fibonacci( $f_0, f_1, n$ : in integer; fibo: out integer) is
   $f_{n1}, f_{n2}, f_n$ : integer;
   $i$ : integer;
begin
  if ( $n > 1$ ) then
     $f_{n1} := f_0$ ;
     $f_{n2} := f_1$ ;
     $i := 2$ ;
    while ( $i \leq n$ ) loop
       $f_n := f_{n1} + f_{n2}$ ;
       $f_{n1} := f_{n2}$ ;
       $f_{n2} := f_n$ ;
       $i := i + 1$ ;
    end loop;
  else
     $f_n := -1$ ;
  end if;
  fibo :=  $f_n$ ;
end fibonacci;

```

6.5.1 Static Relations

For the fibonacci program:

$MOD = \{fibo, f_n, f_{n1}, f_{n2}, i\}$, and $PRE = \{n, f_0, f_1\}$

$S_V = \{(1, n), (2, f_0), (3, f_1), (5, n), (6, f_0), (6, f_1), (7, f_0), (7, f_1), (8, f_0), (8, f_1), (11, f_0), (11, f_1), (11, n)\}$

$V_S = \{(fibo, 1), (fibo, 2), (fibo, 3), (fibo, 4), (fibo, 5), (fibo, 6), (fibo, 7), (fibo, 8), (fibo, 9), (fibo, 10), (i, 1), (i, 4), (i, 5), (i, 9), (f_n, 1), (f_n, 2), (f_n, 3), (f_n, 4), (f_n, 5), (f_n, 6), (f_n, 7), (f_n, 8), (f_n, 9), (f_n, 10)\}$

$\{(f_{n1}, 1), (f_{n1}, 2), (f_{n1}, 3), (f_{n1}, 4), (f_{n1}, 5), (f_{n1}, 6), (f_{n1}, 7), (f_{n1}, 8), (f_{n1}, 9), (f_{n2}, 1), (f_{n2}, 2), (f_{n2}, 3), (f_{n2}, 4), (f_{n2}, 5), (f_{n2}, 6), (f_{n2}, 7), (f_{n2}, 8), (f_{n2}, 9)\}$

$V_i = \{(fibo, f_0), (fibo, f_1), (fibo, n), (f_n, f_0), (f_n, f_1), (f_n, n), (f_{n1}, f_0), (f_{n1}, f_1), (f_{n1}, n), (f_{n2}, f_0), (f_{n2}, f_1), (f_{n2}, n), (i, n), (i, i), (n, n), (f_0, f_0), (f_1, f_1), (f_{n1}, f_{n1}), (f_{n2}, f_{n2})\}$

These relations explicitly specify that n, f_0 and f_1 are input variables and are not modified by the program. The relation V_i can be used to generate program projections. For example,

$P_{fibo}^i = \{1, 4, 5, 9\}$ that represents the program:

```

if ( $n > 1$ ) then
   $i := 2$ ;
  while ( $i \leq n$ ) loop
     $i := i + 1$ ;
  end loop;
end if;

```

6.5.2 Dynamic Relations

Suppose the fibonacci program is being executed for $n = 1$. The dependence relations for the program are more precise when the program execution information is available to the dependence relations generator. For this input value, statements 1, 10 and 11 will be executed and $MOD = \{f_n, fibo\}$, and $PRE = V - MOD$. The relations are:

$S_V = \{(1, n), (11, n)\}$.

$V_S = \{(fibo, 1), (fibo, 10), (fibo, 11), (f_n, 1), (f_n, 10)\}$, and

$V_i = \{(fibo, n), (f_n, n), (f_0, f_0), (f_1, f_1), (n, n), (f_{n1}, f_{n1}), (f_{n2}, f_{n2}), (i, i)\}$.

For this execution, $P_{fibo}^{fibo} = \{1, 10, 11\}$, corresponding to the program:

```

if ( $n > 1$ ) then
  null;
else
   $f_n := -1$ ;
end if;
fibo :=  $f_n$ ;

```

While debugging, if the output value of the variable fibo is erroneous then only three statements are to be considered for finding the cause of the error.

7 Visibility and Control Flow Graphs for Ada

Using the formalism developed in earlier sections, the language dependent definitions can be provided for Ada. Ada uses linear elaboration, implicit and explicit nesting, and packages with special visibility export and import mechanisms. For example, a declarative context in Ada can be: specification and body of procedures, functions and packages, and the block statement. With in a declarative context, the units correspond to: variable, type and constant declarations; assignment, if and loop statements; procedure calls; and expressions (including variables and function calls).

7.1 Visibility Flow Graphs

Suppose a declarative context M directly nests N after unit $i-1$ where N is not a package. Nodes M and N , and the arc (M, N) are added

Unit	R_i	D_i
type e_1 is new e_2	$\{e_2\}$	$\{e_1\}$
$e_1 : e_2$	$\{e_2\}$	$\{e_1\}$
function $e \dots$	ϕ	$\{e\}$
procedure $e \dots$	ϕ	$\{e\}$
package $e \dots$	ϕ	$\{e\}$
$e \equiv \text{expr}$	$\{e\} \cup E_{\text{expr}}$	ϕ
if (expr) then...	E_{if}	ϕ
while (expr) loop...	E_{while}	ϕ
	$E_{\text{expr}} \equiv \{ \text{if expr contains } \}$	

Table 1: Units for Ada Declarative Contexts

Statement	CFG Tree
$e \equiv \text{expr}$	assignment (e, expr)
while (expr) loop statements	while (expr, statements)
if (expr) then S_1 else S_2 end statements	if (expr, S_1 , S_2)
e (procedure call)	sequence(S_1 , $S_2 \dots S_n$)
expr	call (e)
node (child ₁ , child ₂ , ..., child _n)	expr (expree)
	node (child ₁ , child ₂ , ..., child _n) is a CFG node with n children

Table 2: Primitives for Ada Control Flow Graphs

to the VFG. The function $OUT_{(M,N)} \equiv V_j^M$ and $IN_{(M,N)} \equiv i$. As there is no visibility flow from the context N to M , arc (N,M) is not added to the graph.

For explicit nesting, using *separate* and *is separate* clauses, the definitions for IN and OUT are the same. However, the connection of the nodes N and M with the arc (M,N) is postponed to the linking phase when individual VFGs, corresponding to different library units, are linked together.

If the context N is a package then there is an arc (N,M) denoting the flow of visibility from the public part of the package N to its parent M . If j is the last unit in the public part, then $OUT_{(N,M)} \equiv V_j^N$. At the target of this arc, $IN_{(N,M)} \equiv i$.

The packages can be used across the library units using the with clause. If entity M uses the package N with a with clause, the associated visibility flow is represented by the arc (N,M) . The IN and OUT relations are the same as for the arc (N,M) as described before for M nesting the package N . As for explicit nesting, arcs corresponding to the with clause are set up for the VFGs corresponding to all library modules have been connected. The sets D_i and R_i are defined for the various types of units in table 1.

7.2 Control Flow Graph

Control flow graphs for Ada programs with if and loop statements are defined using the primitives displayed in table 2.

8 Implementation of an Ada Program Analyzer

Currently we are in the process of implementing an Ada Program Analyzer (AdaAn). Here the design of AdaAn is outlined, and the functionality of its main components is discussed. The system is be-

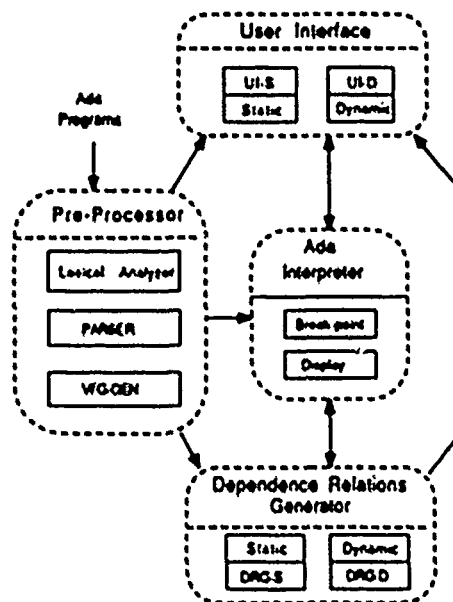


Figure 1: AdaAn: An Ada Program Analyzer

ing implemented on Sun workstations using Unix²/C platform and tools like Lex, a lexical analyzer generator and Yacc, an LALR parser generator.

8.1 VFG Generator

The VFG for a program is generated in a bottom-up fashion. The grammar of Ada is described in the Yacc notation. Currently only a representative subset of Ada has been targeted.

Attributes can be defined for each node of the parse tree. For a VFG, this is used to generate the unit-list and information for the VFG edges. At the expression level, the R sets are the only attributes as the D sets are always empty. Whenever a production corresponding to a declarative context reduces, the unit-list is ready and can be attached to the header (newly created for the recently identified context). While analyzing a compilation unit, the intra-unit VFG arcs are fully determined.

The VFG-GEN component takes a program, generates the VFG, and when all the programs have been transformed, links the individual VFGs together. The linking phase provides resolution for all the unresolved names in *is separate* and *with* clauses. An *is separate* clause requires a corresponding *separate* clause in a subunit. Similarly, the *with* clause needs a package that is implemented as a separate compilation unit. If the package specification and the corresponding body are implemented as different compilation units (the specification as a library unit and the body as a secondary unit), they are also linked together with a VFG arc.

After the linking phase, there are two types of VFG arcs. The arc *A_NEST* corresponds to traditional nesting in block structured languages and signifies a uni-directional flow of visibility from the parent to its child. To accommodate packages, the arc *A_PACK* links a package to another declarative context describing the visibility of the package name, and eventually the non private declarations of that package.

²Unix is a trademark of AT&T

8.1.1 Internal and External Data Structure for VFGs

Internally, the VFGs are represented using pointers and structures. Each declarative context corresponds to a structure with fields to store its ID, name, fullname, filename, begin_line, end_line and number_of_units. A list is maintained for both incoming and outgoing VFG areas.

The VFG representation can be transformed for disk storage. A Lisp like notation is used with balanced parenthesized lists. This representation will be useful for the incremental analysis of large programs where only the modified source files are reproduced.

8.2 Dependence Relations Generator

Three units together comprise the dependence relations generator. Dependence relations for assignment statements can be generated directly from the syntax tree of the programs. For compound statements, information about the statement types, and structure of expressions can be prepackaged. This is performed by the *Pre-Processor*, and its output is used to generate relations in both static and dynamic contexts. In the static situation, *DRG-S* (Dependence Relations Generator in Static Context) requires information only from the preprocessing phase. However, for *DRG-D*, information from the Ada interpreter is also needed so that the actual program trajectory is utilized for generating dependence relations.

8.3 User Interface

The functionality of the user interface is divided into two parts dealing with the static and dynamic aspects of the program semantics. *UI-S* (User Interface for Static Semantics) implements a browsing tool supporting cross-referencing, visibility and scoping properties, and information generated from the static dependence relations.

The *UI-D* component handles the dynamic semantics of programs. Traditional debugging can be supported by supplementing the interpreter with a break-point and display facility. This debugging process is improved by using the dependence relations generated by *DRG-D*.

9 Program Maintenance

The primary motivation for this work is to facilitate the development of analysis tools for supporting program maintenance. In the maintenance phase, a smaller number of programmers are available. The original development of the program may have involved more individuals for developing different parts of the program. Therefore, effective decomposition schemes and suitable browsing tools, to facilitate comprehension, are more important for program maintenance. Support for debugging and verification is also needed.

9.1 Program Comprehension

Cross reference generators are useful to extract information from program texts. The VFG based interface produces this information in an on-demand fashion and portrays the full visibility rules of the language. For large programs, this is more effective compared to a comprehensive listing of the cross reference information. Furthermore, other types of secondary information, arising from the transitive nature of many relationships, are not directly present in the listing. On the other hand, the interactive interface is ideally suited for these special types of queries, without inundating the user with unrequested information.

9.2 Program Debugging

Program slicing [12] was shown to be useful for program debugging as one can automatically identify only those program statements that may have been used for determining the current value of the variable under consideration. Partial statements [3] and static program projections are similarly useful for this type of debugging.

In the PELAS [7] system, both static and dynamic analyses have been employed to produce dependence networks that can identify the statements whose execution results in the current (and erroneous) value of the variable under consideration. The system maintains the entire trajectory of the program; a single statement, if executed several times, may correspond to multiple nodes in the trajectory. The dynamic program projections also utilize the program's trajectory but maintain the analysis information as dependence relations of fixed size. However, they are more precise than static projections or program slices.

9.3 Program Testing

It is possible to utilize the dependence relations to simplify program testing associated with maintenance. Suppose the output value of a variable v is to be studied while the program is being tested. This variable could be a new one, just recently included in the program during maintenance. From the static dependence relation V_r , one can determine all the variables such that the output value of v depends on the input value of these variables. Only these variables need effective input values and the corresponding program projection can be executed for debugging purpose. Thus, only the relevant portion of the program requires to be tested.

9.4 Formal Verification

Dependence relations may also be useful in formal proofs. Suppose it is required to prove a particular assertion that involves only some of the program variables (may be those which are recently introduced during program maintenance). Instead of considering the effect of the entire program on initial assertions, the dependence relations can be used to identify all input variables and statements on which the assertion to be proved depends. This could especially be effective during program maintenance, where a procedure is modified to add a functionality that is entirely unrelated to the old functionality of the procedure. A procedure may contain two groups of statements that do not interdepend on each other. Such modifications are typical in the maintenance phase.

10 Concluding Remarks

Our work is aimed towards the identification of decomposition schemes applicable towards the static and dynamic analyses of programs. We provide the visibility flow graphs as an intermediate representation form for static analysis of multi-module programs. Relations are defined to capture the interdependence among variables and statements of a program in both static and dynamic contexts. These relations are used to identify and generate program projections, and other useful information for debugging, testing, and program-proofs. A prototype analyzer, called *AdaAn* has been designed that incorporates these schemes for analyzing Ada programs.

The implementation of *AdaM* is continuing. Most of the components of the *Pre-Processor* including the lexical analyzer, the parser, and the *VFG* generator have been implemented. Algorithms for generating the dependence relations for static and dynamic situations have been developed and are currently being implemented. The user interface is partially completed so that the *VFG* information can be displayed. On the theoretical side, the usefulness of the dependence relations for testing and proving programs is being further explored and formalized.

We envision that the current work can be extended along the following lines.

- Augmenting the *AdaM* functionality with support for test and proof maintenance
- Implementing an analyzer for the full Ada language with arbitrary control flow (goto's), exceptions, recursion and tasking.
- Using dependence relations to restructure the existing programs for simplifying future maintenance.
- Extending this approach for analyzing parallel programs.
- Verifying the scope and effectiveness of such an analyzer for program maintenance in production environments.

Acknowledgements

I am grateful to Dr. S. R. Schach for his continuous guidance and encouragement for this project.

Special thanks are due to Heena Prasad for assisting me with graphics and textprocessing.

References

- [1] *Reference Manual for the Ada Programming Language*. United States Department of Defense, 1983.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Code optimization. In *Compilers: Principles Techniques and Tools*, chapter 10, pages 585-722. Addison-Wesley Publishing Company, 1986.
- [3] Jean-Francois Bergeretti and Bernard A. Carre. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37-61, January 1985.
- [4] B. W. Boehm. Improving software productivity. *Computer*, 43-58, September 1987.
- [5] Rajeev Gopal. On supporting software evolution. April 1989.
- [6] Rajeev Gopal. On supporting software evolution—decomposition schemes for static and dynamic analysis of programs. 1989.
- [7] B. Korel. PELAS—program error-locating assistant system. *IEEE Transactions on Software Engineering*, SE-14(9):1253-1260, September 1988.
- [8] K. Narayanaswamy and W. Scacchi. Maintaining configurations of evolving software systems. *IEEE Transactions on Software Engineering*, SE-13(3):324-334, March 1987.
- [9] Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6:307-334, 1986.
- [10] David A. Schmidt. Languages with contexts. In *Denotational Semantics*, chapter 7, pages 137-173, Allyn and Bacon Inc., 1986.
- [11] N. F. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):303-310, March 1987.
- [12] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352-357, July 1984.
- [13] Alexander L. Wolf, Lori A. Clarke, and Jack C. Widlen. A model of visibility control. *IEEE Transactions on Software Engineering*, SE-14(4):512-520, April 1988.

Author



Rajeev Gopal received the B.E.(Hons.) degree in Electrical Engineering, and the M.Sc.(Hons.) degree in Physics from the Birla Institute of Technology and Science, Pilani, India, in 1983. He also received the M.S. degree in Computer Science from the Vanderbilt University in 1986.

He is a Ph.D. candidate in the Department of Computer Science, Vanderbilt University. His dissertation research is on the design of automatic static and dynamic program decomposition techniques for supporting software evolution. His other research interests include programming languages and systems, programming environments, search methods in artificial intelligence and machine learning, and genetic algorithms.

During 1986-87, he was affiliated with the lab-automation projects for cancer and AIDS drug development programs of the National Cancer Institute. He served as a systems analyst and subsequently the department manager for the In Vitro Screening System Group of VSE Corporation, Alexandria, VA.

Mr. Gopal is a student member of the Association of Computing Machinery and the IEEE Computer Society.

Address

email: rjg@vuse.vanderbilt.edu
Department of Computer Science
Vanderbilt University
Box 5632, Station B
Nashville, TN 37235

An Object-Oriented Approach to Simulating a Real-Time System in Ada

Johan Margono and James E. Walker

Network Solutions, Inc
8229 Boone Boulevard
Vienna, Virginia 22180

Abstract

A real-time embedded system is typically characterized by its' ability to dynamically change the priorities of its' concurrently executing processes. Furthermore, processes with high priorities must be selected to execute in favor of those with lower priorities to ensure that system requirements have been met. Ada does not directly support dynamic priorities in that Ada tasks are assigned static priorities at compilation time. However, we will illustrate how to utilize Ada's tasking features to provide for dynamic priorities.

It is a proven fact that Ada's definition for priority scheduling has severe deficiencies when utilized in hard real-time environments. Nevertheless, there is nothing prohibiting the software engineer from implementing his/her own scheduling algorithm. As a case study we developed an application in Ada that simulates a simple operating system. Our approach not only supports dynamic priorities but also provides an innovative way to handling the deficiencies in priority scheduling.

The purpose of this experiment was: 1) To provide an innovative approach to simulating a simple real-time system in Ada; 2) To use Ada tasking for implementing a dynamic priority task scheduling algorithm; 3) To demonstrate the feasibility of using an object-oriented approach for developing real-time systems; 4) To dispel the belief that Ada supports only one task scheduling algorithm (i.e., The one supplied by the compiler vendor).

Ada and Real-Time Systems

Many Ada advocates have voiced their concerns on the problematic features of Ada in real-time applications. Needless to say, these concerns are mostly focused on the inadequacy of Ada in addressing critical timing issues in applications where meeting most, if not all, deadlines are probably the most paramount aspects.

Most experts believe that the "real time scheduling work ... is concerned with separating timing issues from logical correctness issues in real-time system design. In current practice, these issues are intertwined to the extent that program structure is dominated by timing issues rather than issues of modifiability, maintainability, understandability, etc."¹

An Object-Oriented Approach

An Object-Oriented Design (OOD) methodology was used to develop the simulator, rather than the traditional top-down functional decomposition approach. Realizing that Ada is not an object-oriented programming language, we felt that we could view the problem space from a more realistic perspective as a collection of objects and associated operations. Each module in the software system denotes an object (abstract state machine) or class of objects (abstract data type). Each object in our operating system simulator was implemented as an Ada task. Depending on the "functionality" of the object. It could be implemented as a server task, an actor task, or an agent task. For instance, a

¹ Goodenough, J. B. Real-Time Scheduling, TRI-Ada '88 Proceedings, p. 134.3

CPU object is considered as a server task because it does not "act" on other task objects. A process, on the other hand, is an actor task because it must periodically make a request for service.

Operating System Simulation²

As mentioned before, the experiment was the implementation of a simple operating system simulator. The simulator was to provide at least the following: 1) Allow for dynamic priorities; 2) Allow for multiple RAM's, CPU's, IOP's, Disks, and Files; 3) Allow simultaneous disk access; 4) Allow multiple Read Access of files.

A process is initially assigned a priority. Depending on how busy the CPU is, a process may have to spend a lot of time waiting for the CPU. Ada's FIFO task entry scheduling is not adequate to prevent starvation. Thus, another approach was taken. First, the longer a process waits for a CPU (or other resources), the higher its' priority becomes (also known as "aging"). After a service has been granted, the process's priority will have to be reset to its' initial priority to prevent it from indefinitely postponing other processes for services. Second, "Families of Entries" were used to implement a task scheduling algorithm which is not strictly FIFO. Because a task entry has its' own service queue, a family of entries in effect is a very close approximation of a multiple feedback queuing approach. This approach allows a process with a higher priority to be serviced prior to any other processes with a lower priority even though the higher priority process arrived at a later time.

Our operating system consists of the following resources: Central Processing Units (CPUs), Random-Access Memories (RAMs), Input-Output Processors (IOPs), Secondary Storages (Disks), and Disk Files (Files).

In this simulation, we adopt the following resource allocation rules:

- i. Resources are allocated based on the priority of the requesting process. High-priority processes will always be preferred over low-priority processes. If two processes of the same priority are competing for a resource simultaneously, then the choice becomes arbitrary.
- ii. *RAMs*: Each RAM will have ten equally sized slots for processes. All processes occupy the same amount of RAM space. There is no RAM preemption. Once a process gains access to a RAM, it will remain there until completion. Further, processes do not do dynamic memory allocation.
- iii. *CPUs*: The number of CPUs is allowed to vary. We ran the simulator with one, two, three, four, and five CPUs. CPU preemption takes place after a process has been running for ten consecutive time units.
- iv. *IOPs*: An IOP manages accesses to a disk. Before a process can access a file on a disk, it must first gain access to an IOP. More than one I/O operation can occur at the same time, but not to the same disk. IOP preemption is not allowed.
- v. *Disks*: Our operating system consists of four disks, each of which contains four disk files.
- vi. *Files*: A file is identified by its name with the following convention: *FileNumber-DiskNumber*, e.g., file #1 on disk #3 is named: *F1D3*. A process may have the following types of accesses to a file: read-only or write-only.

² A complete source code listing of the simulation is available upon written request addressed to either author.

Once a process gains access to a RAM, it performs two types of requests: CPU and IO. A CPU request is always followed by an IO request, and vice-versa. The length of each activity is randomly generated. For simplicity (and to avoid polling), each process is assigned to specific CPU and IOP. A *fair* algorithm is used to assign equal number of processes to each CPU and IOP. External events and interrupts are simulated by using various rendezvous mechanisms.

Process Life-Cycle

When a process arrives into the system, it must then compete for a RAM space. (In order for a process to be able to run, it must reside in core.) Since we only have one RAM which will only hold ten processes, some processes will likely have to wait for a RAM space. Once a process is in core, then it can be in one of the following states:

- i. *Running*. A process is running once it has gained access to its assigned CPU.
- ii. *Blocked for an IOP*. A process is blocked for an IOP if it is requesting to do an I/O and there are no available IOPs.
- iii. *Blocked doing IIO*. A process is blocked doing I/O after it has gained access to an IOP, the disk where the file resides, and to the file itself.
- iv. *Ready*. A process is in a ready state when it is waiting to gain access to a CPU. A process can be in a ready state after it has gained access to a RAM space, after its time-slice has expired, or after completing an I/O burst.

Since an object-oriented approach is used, the traditional concept of a scheduler, or cyclic executive, is absent from our simulator. A process is an intelligent, yet obedient, object. On one hand, it knows when to wait for a CPU, when to access a file, or when to release an IOP. On the other hand, it always abides by the rules given earlier.

Process Class

Traditionally, a scheduler would be responsible for creating, dispatching, and terminating processes. In our design, however, processes are represented as Ada tasks which will communicate directly with resources such as CPUs, RAMs, IOPs, and so on. CPU and I/O bursts are simulated by using a delay statement. Although a delay statement is known to be an inaccurate timing mechanism, it is considered to be appropriate for our solution because of portability reasons. Finer resolution timing can be used instead if the environment provides an external clock and interrupt.

```

...
package Process is
  type State is
    (Blocked_For_File,
     Blocked_For_IO,
     Blocked_Doining_IO,
     Ready,
     Running);
  ...
  task type Class is
    ...
    entry Start;
    entry Stop;
    ...
  end Class;
  ...
end Process;

...

package body Process is
  ...
  task body Class is
    ...
  begin -- Class
    ...
    accept Start;
    -- synchronize with the kernel
    -- The_Memory.

```

```

    Seize(The_Priority);
    Until_No_More_Requests:
for Index in The_Requests'Range
loop
    case The_Requests(Index).
        Kind is
        when
            Request.CPU_Request =>
                ...
            -- simulate round-robin
                ...
        when
            Request.IO_Open_Read_Request
                ...
        when
            Request.IO_Open_Write_Request
                ...
        when
            ...
        when
            Request.IO_Close_Request =>
                ...
        end case;
    end loop
    The_Memory.Release;
    accept Stop;
    -- synchronize with the kernel
    ...
end Class;
...
end Process;

```

We chose to implement Process in a *round robin* (RR) fashion. This scheduling discipline remedies the somewhat indiscriminate behavior of the *First-In First-Out* (FIFO) strategy. The basic idea of an RR scheduling discipline is that the server (in our case, Process) is allocated to a job only for a time quantum of fixed length (in this case, ten time units). When a job runs out of its time slice (quantum), it cycles back to the "rear of the job queue" to wait for another time slice. In our implementation of Process, this is simulated by using a delay statement, a simple rendezvous, and a loop statement.

An RR discipline, unlike FIFO can only be applied to a preemptive resource such as a CPU, not to a non-preemptive resource such as, in the simulation, the RAMs.

Processor Class

A processor class, which is the parent class of CPUs and IOPs, is implemented as a task with multiple entries, one of which is a family of entries. Entries Start and Stop are used only for synchronization points with the kernel.

```

with Priority;
package Processor is
    type State is (Idle, Running);
    ...
    task type Class is
        ...
        entry Start;
        entry Seize (Priority.Class);
            -- seizing a processor is based
            -- on the priority of the
            -- High-priority processes have
            -- a better chance of seizing.
        entry Release;
        entry Stop;
        ...
    end Class;
    ...
end Processor;
...

package body Processor is
    task body Class is
        ...
        begin -- Class
            ...
            accept Start;
            -- synchronize with the kernel
            Forever:
            loop
                ...
                select
                    accept Seize(9);
                or
                    when (Seize(9)'Count = 0) =>
                        accept Seize(8);
                or
                    when (Seize(9)'Count = 0) and
                        (Seize(8)'Count = 0) =>
                            accept Seize(7);
                ...
                or
                    when (Seize(9)'Count = 0) and
                        (Seize(8)'Count = 0) and
                        (Seize(7)'Count = 0) and

```

```

        (Seize(6)'Count = 0) and
        (Seize(5)'Count = 0) and
        (Seize(4)'Count = 0) and
        (Seize(3)'Count = 0) and
        (Seize(2)'Count = 0) =>
        accept Seize(1);
    or
    when (Seize(9)'Count = 0) and
        (Seize(8)'Count = 0) and
        (Seize(7)'Count = 0) and
        (Seize(6)'Count = 0) and
        (Seize(5)'Count = 0) and
        (Seize(4)'Count = 0) and
        (Seize(3)'Count = 0) and
        (Seize(2)'Count = 0) and
        (Seize(1)'Count = 0) =>
        accept Stop;
        -- stop only when there are
        -- no more outstanding
        ...
        exit Forever;
    end select;
end loop Forever;
...
end Class;
end Processor;

```

The select statement in the Processor body is constructed such that a high-priority task that tries to seize is always given a chance to do so. Other tasks with lower priorities must wait in their respective entry queues. Note that for this to occur, these rendezvous attempts must arrive at their entry points "simultaneously." If a low-priority task arrives prior to a high-priority task, then the low-priority task will always be given access.

Ada's rendezvous mechanism is strictly FIFO. If several client tasks wish to rendezvous with the same entry belonging to a server task, then they are serviced depending on the order of arrival (regardless of task priorities). Notice that this approach avoids starvation.

In a real-time system, however, the goal is not to avoid starvation, but rather, to avoid missing deadlines (given time and space constraints). Our implementation of the Process class above illustrates an example whereby entry families are used to achieve the effects of different priority algorithms for entry queues.

Disk Class

We stated earlier that a Disk object cannot be simultaneously accessed by more than one process. Therefore, the Disk class is simply implemented as a *binary semaphore*. The danger of using semaphores is obvious. A *seize* operation must always be accompanied by a *release* operation. Failure to do so has been known to cause a deadlock. (Euphemistically called *deadly embrace*!)

```

package Disk is
    task type Class is
        entry Seize;
        entry Release;
    end Class;
end Disk;

...

package body Disk is
    task body Class is
        In_Use : Boolean := False;
    begin -- Class
        Forever:
        loop
            select
                when not In_Use =>
                    accept Seize;
                    In_Use := True;
            or
                when In_Use =>
                    -- to prevent cheating
                    accept Release;
                    In_Use := False;
            or
                terminate;
            end select;
        end loop Forever;
    end Class;
end Disk;

```

File Class

Multiple access to a File object is permitted, therefore we implemented File class as a *counting semaphore*, instead of binary semaphore.

```

package File is
    type Operation is (Read, Write);
    task type Class is
        entry Seize (Mode : in

```



```

    entry Release (Operation);
  end Class;
end File;

...

package body File is
  task body Class is
    Number_Of_Readers : Natural := 0;
  begin -- Class
    Forever:
    loop
      select
        accept Seize (Mode : in
          do
            case Mode is
              when Read =>
                Number_Of_Readers :=
                  Number_Of_Readers
                    + 1;
              when Write =>
                while Number_Of_Readers
                  loop
                    accept Release(Read);
                    Number_Of_Readers :=
                      Number_Of_Readers
                        - 1;
                  end loop;
              end select;
            end Seize;
            if Number_Of_Readers = 0 then
              accept Release(Write);
            end if;
          or
            accept Release(Read);
            Number_Of_Readers :=
              Number_Of_Readers + 1;
          or
            terminate;
          end select;
        end loop Forever;
      end Class;
    end File;

```

Notice that in "accept Seize ... end Seize", we check whether a request is to Read or to Write. If it is Read, then we simply increment the Number_Of_Readers. However, when the request is Write, we will only accept Release associated with Read until all current readers relinquish accesses.

Conclusion

Ada has brought about unprecedented challenges in various fields of software engineering in general, and real-time systems in particular. Object-oriented approaches are meant to overcome some, if not most, of these challenges. However, knowing which Ada features are best applied to solving stringent real-time requirements quite often presents a unique and yet familiar problem.

In this paper, we have shown several techniques which can be used to overcome restrictions of Ada tasking. We are aware that current Ada tasking implementations are in general, short from expectations. Nevertheless, we believe that, in order for Ada to succeed, we must proceed with whatever Ada has to offer in solving real-time problems. After all, "a journey of a thousand miles begins with a footstep."

References

- Booch, G. *Software Engineering with Ada*, Benjamin-Cummings, 2nd Edition, Menlo Park, California, 1987.
- Deitel, H. M., *An Introduction to Operating Systems*, Addison-Wesley, Reading, Massachusetts, 1984.
- EVB Software Engineering, Inc., *An Object-Oriented Development Handbook*, to be published by Benjamin-Cummings, 1989.

Ada Implementation of Operating System Dependent Features

Michael I. Schwartz
Randall W. Hay

Martin Marietta Information & Communications Systems

ABSTRACT

Many Ada implementation problems occur because of contention between Ada semantics and those of the underlying operating system. One such problem is that the Ada tasking model is generally implemented inside of a single operating system process. This causes run-time conflict between the Ada run-time task scheduler and the operating system process scheduler, and conceptual conflict between the Ada idea of task state and the operating system idea of process state.

These conflicts lead to operating system dependent designs and solutions, and a body of code that can be as hard to understand as it is to maintain. Solutions of this nature are destructive of the central Ada paradigm.

An alternative exists: to use the features of Ada to mask the operating system dependent parts; to treat the operating system process like any other resource in need of protection; to write packages that are both easy and intuitive to use.

We have taken as an example a real world problem dealing with time-sliced tasking and networked inter-process communications under the UNIX[†] operating system. The specification of *Socket_IO* is virtually identical to that of the standard *Sequential_IO* package, and avoids problems of system scheduling conflicts within multitasking Ada programs. As a bonus, the utility also operates very efficiently.

1. Problem

Ada contains features, such as tasking, that other languages leave to the operating system. The Ada tasking model requires close cooperation between Ada tasks; the cooperation is so close, in fact, that current implementations of Ada under general-purpose operating systems implement all Ada tasks within one operating system process. Ada run-time systems, then, provide their own task schedulers to determine which Ada task may run during the time-slice that the operating system scheduler gives to the entire Ada-based operating system process.

The Ada scheduler must maintain the state of the Ada tasks (e.g., *AWAITING_RENDEZVOUS*, *RUNNABLE*), and the operating system must maintain the state of the Ada-based process (e.g., *AWAITING_SIGNAL*, *RUNNABLE*). Doing useful work in a program usually involves making calls to the operating system. A task making a system call runs the risk of putting the entire operating system process in a non-runnable state, rather than simply changing its state in the Ada scheduler. For instance, a *read* call will put the UNIX process in *AWAITING_SIGNAL* state, rather than just telling the Ada scheduler that the task is in *AWAITING_RENDEZVOUS* state.

These problems are typical of conflicts between competing schedulers and competing ideas of process state.

When confronted with these problems, designers and programmers must choose an operating system dependent solution. Operating system dependent code is both hard to understand and difficult to maintain. The

intermixing of system independent operational code with operating system dependent code runs against the grain of the Ada paradigm, making portability and maintainability more difficult because of the implementation language.

Clearly, a better design solution must exist.

2. Background

While considering the issues of operating-system dependent code, we encountered a problem in a multi-tasking Ada program doing network communications. Each time a task performed a network *read*, the entire Ada-based process would halt. Since the Ada scheduler for our compiler was time-slicing, when the next timer expiration signal was delivered, the *read* was interrupted and tasks were rescheduled.^{*} Each *read*, then, had to check its return status and determine whether it needed to reissue itself. This caused a performance loss because the entire operating system time-slice was lost whenever a task issuing a *read* was scheduled. Ideally, such a task should not have been scheduled until its *read* could complete. The alternative was to prevent task-switching during a *read*. This is unacceptable because there is no way of knowing when, if ever, more data will be available to read. Of course, the operating system scheduler could not know about the task (since UNIX is single threaded), and the Ada run-time scheduler could not know about the *read* call (since the *read* call was a system interface and not an Ada package call).

3. Approach

Our approach was "classical" in the Ada sense. We used standard design techniques to create the overall structure of the *Socket_IO* utility, which included four packages: *Socket_IO*, the user-level package; *Socket*, the system implementation of sockets; *Descriptor*, the abstract data type package; and *System_Interface*, which collected system-specific code including *Pragma Interface* code.

In that light, we viewed the operating system process as the resource needing protection. The Ada technique for dealing with a resource needing protection is to create a *monitor* task to arbitrate all access to that resource.[‡] Our *Socket* package contains a *Dispatcher* task that arbitrates all *read* calls, and converts them into rendezvous. This approach also has the effect of letting the Ada scheduler know that the task is not ready to run. Figure 1 shows conceptually how this mechanism looks.

Centralizing control of system calls allows the Ada scheduler to register them, but it does not yet prevent the entire Ada program from being suspended at the operating system level when, say, a *read* call is made. The way to prevent this is operating system specific.

Under the Berkeley flavor of UNIX, I/O operations can be accomplished asynchronously (i.e., an I/O operation will send a signal when it can complete) as well as synchronously (i.e., the process will pause until the operation completes). It was apparent that to accomplish our goal,

^{*} Ada semantics for task scheduling permit many forms for the scheduler. Many compilers time-slice tasks — that is, each task runs until it blocks or the timer expires. Other compilers simply let each task run until it blocks. Some compilers allow the programmer to change the scheduler's behavior at compilation time or at run time.

[‡] Software Components with Ada, Ch. 13.3, p. 433.

[†] UNIX is a trademark of Bell Laboratories.

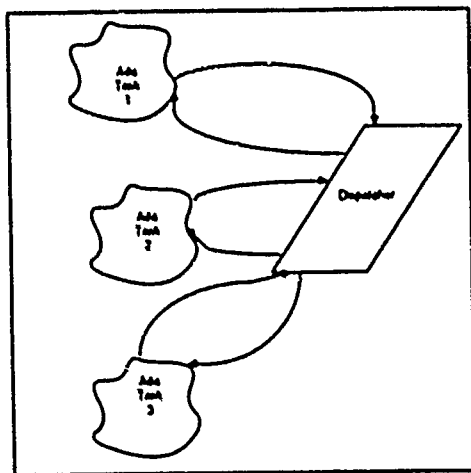


Figure 1: Dispatcher Arbitrating Operating System I/O Calls

asynchronous I/O was the correct mechanism. Asynchronous I/O would not suspend the process; rather when the I/O could complete, our process would be signaled and could complete the rendezvous through *Dispatcher*. Details of this mechanism under Berkeley UNIX are left to the design section. Figure 2 shows the basic interface between *Dispatcher* and the UNIX kernel.

Under DEC's VMS operating system, the same choice of asynchronous I/O would be selected. However, implementing that choice

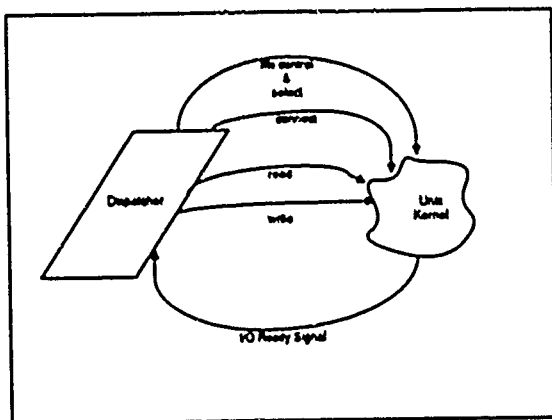


Figure 2: Dispatcher Interface to the UNIX Kernel

would differ considerably. VMS uses a mechanism called Asynchronous System Traps (AST). The VMS mechanism will call a specified subroutine when an operation using ASTs has completed. Under Ada, the mechanism provides a task entry when the operation has completed. Thus, the monitor task would be implemented significantly differently under VMS.

The point is that while the internals of *Dispatcher* would change from operating system to operating system, the programmer-level syntactic and semantic construct of the *read*, *write*, and other calls would be consistent. That is, while the operating system dependent portions of the code would in fact change, the programmers would not have to relearn a new interface for each new operating system.

This leads us into another area of our approach: maintainability. Operating system dependent code is an anathema to Ada thought. How could we properly encapsulate and justify its use in project-oriented Ada development environments?

Our approach to this issue was to observe that implementation of an Ada program on a particular platform under a particular general-purpose operating system assumed *a priori* the existence of personnel with some expertise in that platform and operating system. For convenience, we refer to those people as "Gurus." One traditional problem with depending on Gurus, at least in our organization, is that they tend to be available (albeit heavily loaded) at the beginning of a program, but are reassigned to other programs later in the life cycle. Since Gurus participate in the early development activity, we have defined a specific role for them.

It is a great advantage for non-Guru programmers, whom we call "Gentle Good People," to work with consistent, operating system independent interfaces. Gentle Good People can both program and maintain these interfaces. This makes the job of the Gurus the design (with the system and software designers) and implementation of the operating system dependent parts to provide the proper interface to the Gentle Good People. Gurus will help design and implement packages like *Socket_IO*, while Gentle Good People should find them easy and intuitive to program and maintain.

Wearing our designer's hat in our choice of intuitive interfaces, we selected the *Sequential_IO* package as the best match to what we wanted from our *Socket_IO* package. Any programmer who has used *Sequential_IO* should find the *Socket_IO* package very familiar.

The operating system dependent parts of the code should be the most stable parts of the code, since Gentle Good People are not permitted to program that interface. Should operating system revision, or performance considerations require changes to that code, a Guru should be called in to perform that maintenance.

We feel that in this way the Ada paradigm is best preserved while recognizing the inevitability of operating system dependent code.

4. Design

Standard Ada principles were used in designing four interrelated packages. For convenience, an overview will be provided first. Second, the system independent portions of the design will be presented. Finally, the system dependent portions of the design will be discussed.

4.1. Overall Design

The *Socket_IO* package interface was designed to duplicate the *Sequential_IO* specification with all of the functions and procedures either implemented or stubbed out for future use. This allows users of *Socket_IO* to design their applications to use either package. Software created to this design can use either network interfaces (sockets) or sequential files for communications between processes. This in turn allows the software to be designed and coded before the design decisions of hardware allocation or application location, supporting the "software first" advantages of software engineering.

The principles used in package design were that as much as possible, package specifications were designed to be system independent. This required the design to be informally tested to several available systems. Second, the bodies were classified as maintainable by Gentle Good People or Gurus. We tried to avoid bodies that were classified as needing one type of maintainer except for a procedure or two. The package bodies that were classified as maintainable by Gentle Good People tended to be mostly system independent.

The resulting *Socket_IO* utility consists of four major packages. The generic package *Socket_IO* (see *Sequential_IO*'s specification in the ANSI/MIL-STD-1815A Section 14.2.3 for details of the generic parameter and the allowed functions and procedures), the two packages *Descriptor* and *Socket* which are object abstractions used by *Socket_IO* and the *System_Interface* package which contains the system dependent interfaces to the UNIX operating system. A high-level diagram of the package interfaces is provided in Figure 3.

The *Socket_IO* package is the only package directly accessible by the clients of the utility. Both the specification and body of *Socket_IO* are classified as system independent and maintainable by Gentle Good People.

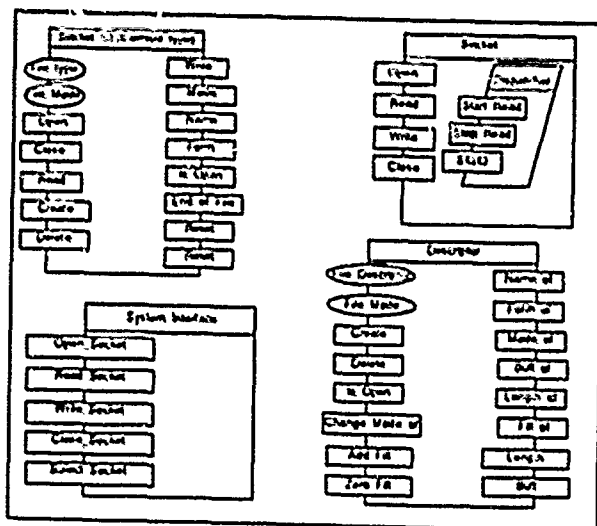


Figure 3: Diagram of Package Design

The *Descriptor* package is an abstraction for the object class descriptor. This class of objects is represented by an internal file of records with a unique *file_descriptor* (*Socket_ID*) as the key for access. There are 14 functions and procedures allowed on this class of objects. The *Descriptor* specification is system independent; but as the information required for proper use of a descriptor may change from system to system, the body is system dependent. The functions are accessor and bookkeeping functions, so the body is maintainable by Gentle Good People.

The *Socket* package is an abstraction for the object class socket. This class of objects represents the actual operating system sockets and allows six operations to be performed on sockets. The *Socket* specification is system independent; since the body contains the dispatcher task, it is highly system dependent. We used the separate feature of Ada to remove the code for the *Dispatcher* body from the *Socket* body. The remainder of the *Socket* body requires a very Ada knowledgeable software engineer to maintain, but not necessarily a Guru.

The *System_Interface* package encapsulates the system calls and UNIX routines to provide maximum portability with the fewest number of modules being affected. The *System_Interface* package consists mostly of PRAGMA INTERFACE declarations. Both the specification and body are very system dependent. We classified this package as requiring a Guru, because classifying it otherwise early on led to programming blunders.

4.2. System Independent Design

System independent design used the object oriented approach. The individual objects (file descriptors, buffers, monitors) were converted into packages and the operations on those objects to functions and procedures. This approach allows for easier maintainability by the Gentle Good People and Gurus and a cleaner transition from host system to host system.

The following parts of packages were classified as system independent: the specifications of *Descriptor*, *Socket_IO*, and *Socket*; the body of *Socket_IO*; also, the specification of the task *Dispatcher* is system independent with the exception of one system dependent task entry point.

4.2.1. The *Socket_IO* Package Design

The package *Socket_IO* was designed to convert the user interface of *Sequential_IO* to the monitor function of *Socket*. The *Socket_IO* package provides the *FILE_TYPE*, *FILE_MODE*, and necessary operations to allow the using package to receive and transmit data to and from other processes. The package has the only knowledge of the users' *element type* from the generic specification. Therefore, it provides the initial conversion from the users' objects to the buffers the system needs to read and write. It establishes the length and type of *element type* for any necessary conversions. The specification for *Socket_IO* is found in Listing 1.

```
with IO_Exceptions;
with Descriptor;
generic
  type ELEMENT_TYPE is private;
package Socket_IO is
  type FILE_TYPE is limited private;
  type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);

  function MODE (FILE : in FILE_TYPE) return FILE_MODE;
  function NAME (FILE : in FILE_TYPE) return STRING;
  function FORM (FILE : in FILE_TYPE) return STRING;
  function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;
  function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;

  procedure CREATE (FILE : in out FILE_TYPE;
    MODE : in FILE_MODE := INOUT_FILE;
    NAME : in STRING;
    FORM : in STRING := "");

  procedure OPEN (FILE : in out FILE_TYPE;
    MODE : in FILE_MODE;
    NAME : in STRING;
    FORM : in STRING := "");

  procedure CLOSE (FILE : in out FILE_TYPE);

  procedure DELETE (FILE : in out FILE_TYPE) renames CLOSE;

  procedure RESET (FILE : in out FILE_TYPE;
    MODE : in FILE_MODE);

  procedure RESET (FILE : in out FILE_TYPE);

  procedure READ (FILE : in FILE_TYPE;
    ITEM : out ELEMENT_TYPE);

  procedure WRITE (FILE : in FILE_TYPE;
    ITEM : in ELEMENT_TYPE);

  NAME_ERROR : exception renames IO_Exceptions.NAME_ERROR;
  USE_ERROR : exception renames IO_Exceptions.USE_ERROR;
  STATUS_ERROR : exception renames IO_Exceptions.STATUS_ERROR;
  MODE_ERROR : exception renames IO_Exceptions.MODE_ERROR;
  DEVICE_ERROR : exception renames IO_Exceptions.DEVICE_ERROR;
  END_ERROR : exception renames IO_Exceptions.END_ERROR;
  DATA_ERROR : exception renames IO_Exceptions.DATA_ERROR;
private
  type FILE_TYPE is new Descriptor.FILE_DESCRIPTOR;
end Socket_IO;
```

Listing 1: The *Socket_IO* Package Specification

4.2.2. The *Socket* Package Design

The *Socket* package provides the bridge between the completely system independent *Socket_IO* and the very system dependent *System_Interface* package. The *Socket* package hides the monitor task *Dispatcher* in its body. The OPEN, CLOSE, READ, and WRITE functions provided by *Socket* are close to those of the underlying operating system, rather than designed to mimic existing specification styles like *Socket_IO*. The specification of the *Socket* Package is found in Listing 2.

```

with System;
with Descriptor;
package Socket is
  function OPEN
    (name : in STRING)
    return Descriptor.FILE_DESCRIPTOR;

  function CLOSE
    (socket : in Descriptor.FILE_DESCRIPTOR)
    return Descriptor.FILE_DESCRIPTOR;

  procedure WRITE
    (socket : in out Descriptor.FILE_DESCRIPTOR;
     buff : in System.ADDRESS;
     buff_length : in INTEGER);

  procedure READ
    (socket : in out Descriptor.FILE_DESCRIPTOR;
     buff : in System.ADDRESS;
     buff_length : in INTEGER);

end Socket;

```

Listing 2: The Socket Package Specification

4.2.2.1. The Dispatcher Task

The monitor task *Dispatcher* provides the process protection required to prevent the Ada environment's UNIX process from locking up. This task uses both the *Socket* and *Descriptor* objects to ensure that there is data available at the socket before it allows the socket read call to be issued. It remembers how many requests for reads/writes have been processed and which sockets have outstanding read requests and which do not.

The *Dispatcher* task is buried in the body of *Socket* to completely hide its existence from the casual programmer. The details of *Dispatcher* are deferred until the system dependent design sections.

4.3. System Dependent Design

System dependent design is divided into issues. The issues were the naming scheme for network addresses, handling of "sockets," which are the underlying Berkeley networking mechanism, compiler dependencies, and the resulting design of the system dependent part.

The following parts of packages were classified as system dependent: the bodies of *Socket*, *Descriptor*, and *Dispatcher*. The specification of *Dispatcher* has one system dependent feature, namely the system-interrupt task entry name.

4.3.1. Network Addressing under Berkeley

Berkeley uses sockets to do networking. A socket is an analogy taken from the telephone company's old-style operator interface. Clients "plug in" their open request into an awaiting server "receptacle." The networking protocols built into the UNIX kernel are Internetwork Protocol/Transmission Control Protocol (IP/TCP), developed under the Defense Advanced Research Projects Agency (DARPA). IP provides network and transport layers, and TCP and the Universal Datagram Protocol (UDP) provide session layers (TCP is a stream-based protocol, UDP is datagram based). A network address consists of three parts over the IP networks. The first part is the IP address of the host. Lookup routines are provided to determine an IP address based on host name. An IP address has the form of four bytes. As commonly represented, the bytes are written as decimal integers separated by periods. The second part of an address is the protocol (normally the session layer protocol). The protocol (e.g., TCP, UDP) is represented as a small integer. Again, library routines are provided to translate a protocol name into this integer. The third part of an address is the socket number to perform the connection on. Sockets are also represented by small integers. Since sockets represent services, 1024 socket numbers are reserved for standard service requests, and can only be allocated by privileged processes (i.e., *superuser* processes under UNIX). The system library provides service name lookup routines that convert the name into the socket number.

An address comprises three parts; each of the three parts can be an integer or a string. How can we properly interface to such a scheme in Ada?

Our answer was to build a single parsable character string providing all the necessary information. The string has three parts, each part separated by a colon. For each part, the determination of whether the part was represented by name or number was done by examining the first character in that part of the string. If the character was a digit, the field was parsed as a number, otherwise as a name. In addition, if no string was provided for the protocol, a default protocol was selected by the UNIX system. Some valid addresses follow in Table 1.

Address string	Meaning
atlas:2000:	Host atlas with socket 2000 and default protocol.
64.16.1.23:telnet:tcp	Host with IP address 64.16.1.23 with Telnet protocol under TCP
pluto:3100:udp	Host pluto with socket 3100 and UDP protocol

Table 1: Sample Addresses for the Socket_IO Facility

4.3.2. Socket Communications Techniques Under Berkeley UNIX

After setting up an initial connection for a socket, standard operating system *read* and *write* calls will work over the socket very similarly to a file. The main difference is that while file access is fast, network access can be slow. Therefore any read call may return fewer characters than anticipated, because of a time constraint on the sending side.

4.3.3. Compiler Issues

The Verdex Ada compiler does time-sliced tasking. To do this under Berkeley UNIX, it must use operating system signals to know that a timer has expired. The UNIX system delivers a software interrupt called SIGALRM to signal timer expiration. As stated in the problem section, this signal will interrupt any pending system operations that can be interrupted (such as a *read* call). What is needed is a preemption control mechanism to prevent the Verdex Ada run-time kernel from interrupting a system call. The Ada Real Time Environment Working Group (ARTEWG) had already understood this necessity and has issued a suggested implementation of a preemption control package for real time systems. We used this same interface under a general purpose operating system with the modified semantic notion that preventing preemption means *within this operating system process*. The preemption control concept is similar to the monitor in that both mechanisms aid in protecting the *operating system process* resource.

4.3.4. The Socket Package Body Design

The package *Socket* was designed to act as the natural bridge between the system independent *Socket_IO* and the highly system dependent *System_Interface* packages. This allows the monitor task *Dispatcher* to be hidden from the user and protects the system dependencies from the Gentle Good People. The *Dispatcher* task divides the *Socket.Read* request into two parts. The first part registers the caller's intent to read, through the *START_READ* entry; the second part rendezvous with *Dispatcher* through the *STOP_READ* entry, remaining blocked until the read has completed (or raised an exception).

The *Dispatcher* task receives all UNIX I/O signals as a task entry, thanks to the compiler implementation (most compilers examined support this feature). The specification of the *Dispatcher* task is described in Listing 3.

```

with System;
with Descriptor;
--
task Dispatcher is
  entry Start_Read (file : in Descriptor.FILE_DESCRIPTOR);
  entry Stop_Read (Descriptor.FILE_DESCRIPTOR'(0) ..
    Descriptor.FILE_DESCRIPTOR'(31) );
  entry SIGIO;
  for SIGIO use at System.ADDRESS'REF (23) ;
end Dispatcher;

```

Listing 3: Task Dispatcher Specification

The UNIX *select* call is used to determine which files now have I/O available. When Dispatcher has received enough input to fulfill a *SocketRead* request by a caller, the rendezvous is accepted, and the caller can proceed. Note that the caller's semantics are exactly as if I/O was synchronous.

To perform its function the *Dispatcher* relies on an array of descriptors, which it accesses through the *Descriptor* package. In the *Descriptor* package, descriptor records have the following format:

```
type descriptor_record is record
  fd : file_descriptor;
  name : file_string;
  mode : file_mode;
  form : file_string;
  buff : System.ADDRESS := null_buff;
  len : INTEGER := 0;
  fill : INTEGER := 0;
end record;
```

The file descriptor component *fd* is the index given to a file or socket by the system when opened. The component *name* is the operating system description of the file; the naming convention is described under system dependent design. The *mode* component has the value *IN_FILE*, *OUT_FILE*, *IN_OUTFILE*, or *closed* (the closed mode is available only for internal use). The *form* component is not in used at this time, but is preserved for compatibility with *Sequential_IO*. The *buff* component is used when reading from or writing to a socket. It holds the system address of the user-supplied buffer. The integer *len* represents the length in bytes of the user-supplied buffer. The *len* component is determined from *Element_Type'size* for whatever *Element_Type* the user has instantiated the package with. The *fill* component maintains the current number of bytes read or written. When *fill* reaches *len*, the read or write is complete, and we can rendezvous with the calling procedure.

The Dispatcher task uses the entries *START_READ* and *STOP_READ* to synchronize the interface with the users. The first entry, *START_READ*, notifies the Dispatcher that a read has been requested on a socket. This entry contains a parameter to indicate which socket (*fd*) is to be read.

The *STOP_READ* entry is actually a family of entries indexed by the file_descriptor. This convention is used to respond to one request's completion without regard to other outstanding requests that may or may not be in the process of being read. This allows dispatcher to accept the rendezvous with a read request as soon as it is completed.

The third entry, *SIGIO*, is used by the system to notify the Dispatcher that I/O is available.

Certain details must be attended to make this scheme be viable. The UNIX *select* call must be used two times other than when the I/O signal is raised. First, when an I/O signal has just been handled, files must be checked, as rapid receipt of several I/O signals on several files may cause some signals to be dropped. Second, whenever we have just registered intent to read, we check whether I/O is possible since we will never respond to an I/O signal on a file without intent registered.

While using the I/O signal increases the number of system calls required, it has the desired effect of eliminating polling and allowing remaining Ada tasks to run unmolested while I/O is not possible.

Currently, the *write*, and *open* user-level calls are not implemented asynchronously. In the case of *write*, asynchronism is probably not required. In the case of *open*, however, a process may pause for up to 30 seconds while awaiting connection (if the requested host is unavailable). While we have not implemented it yet, an asynchronous *connect* is possible in UNIX, using the *select* call.

5. Results

One measure of results is the ease of using the *Socket_IO* utility. A fragment of user-level code is provided in Listing 4.

```
with Test_IO;
with Socket_IO;

procedure Socket_IO_User is

  record : constant string := "%Z% %M% %I% %F%";

  type MY_STRING is new STRING (1..32);
  package My_IO is new Socket_IO (MY_STRING);

  FILE : My_IO.FILE_TYPE;
  MODE : My_IO.FILE_MODE := My_IO.IN_FILE;
  NAME : constant STRING := "adst2000.";
  FORM : constant STRING := "";
  ITEM : MY_STRING;
  -- Other declarations

begin

  -- Other code
  My_IO.Open (FILE, MODE, NAME, FORM);
  -- Other code
  My_IO.Read (FILE, ITEM);
  Test_IO.Put_Line (STRING (ITEM));
  -- Other code
  My_IO.Close (FILE);

exception

  when My_IO.STATUS_ERROR =>
    Test_IO.Put_Line ("STATUS_ERROR raised by Socket_IO");

  -- Other exceptions
end Socket_IO_User;
```

Listing 4: User-Level Code with the *Socket_IO* Utility

Another measure of results is development time. The *Socket_IO* concept was developed in about two person-hours by one language Guru and one Ada Guru. The initial prototype took four person-weeks, two by a Guru and two by Gentle Good People. After initial prototyping, the final interface and design required one more person-week, mostly used by the Guru. The final bodies were filled in in four more person-weeks, almost all of that time allocated to Gentle Good People. In summary, the utility was developed in ten person-weeks, with three person-weeks allocated to Gurus and seven person-weeks allocated to Gentle Good People.

Another measure of results is efficiency of the final code. An Ada procedure was built that started five tasks, each of which made a connection to a server. The UNIX shell (*/bin/csh*) time command was used to determine user and system time to measure resource utilization by the Ada program. Two C programs were written for comparison and timed the same way. The first C program used the UNIX signal utility (much like the Ada program), and the second used blocking I/O calls to perform the same job as the Ada program. The results are summarized in Table 2. In this table, user time is the amount of time spent running in the user process space; system time is the amount of time spent running in the operating system kernel space on the user's behalf. However:

- The amounts measured are small enough, and the variance large enough to consider the sample results equivalent.
- A qualitative judgement by observers of the tests noted that the C programs seemed to provide results immediately on delivery. The Ada program seemed to lag message delivery by a fraction of a second.
- The results are for a particular version of a particular compiler, and do not reflect the capabilities of Ada.

Test	Average User Time	Average System Time
Ada Same	0.00	0.30
Ada Remote	0.05	0.30
C Signal Same	0.00	0.40
C Signal Remote	0.00	0.45
C Blocking Same	0.00	0.30
C Blocking Remote	0.00	0.40

Table 2: Results of Tests of Ada versus C code for Network Access

6. Current and Future Directions

Since the initial utility has been completed, we are looking for ways to improve the *Socket_IO* utility to be usable by a larger number of programs. The extensions being examined are:

- Testing and implementing server-capable code. The correct interface for a server task under Berkeley is a good match for the *Create* procedure of *Sequential_IO* (and thus *Socket_IO*).
- Implementing the same utility under DEC's VMS operating system and Ada compiler. TCI/IP (Wollongong) and DECNET interfaces will be considered.
- Porting the same utility to the Alsys Ada compiler.
- Understanding the issues involved with line-oriented interfaces for clients and servers under Ada. A Telnet client is perceived as a good test example.

7. Summary

We found the need to follow several guidelines in writing Ada code for operating system dependent features:

- Base the programming paradigm on existing packages or standards.
- Make sure that Gentle Good People find the package interfaces intuitive, therefore easy to program and maintain.
- Consider operating system dependent portions of a design as critical lower-level computer system components (CLLCSC), and begin work on them early.
- Make the best use of Gurus that are available mostly at the front end of a project, by identifying the operating system dependent portions of the design and encapsulating their details in packages.

8. Bibliography

- (1) Software Components with Ada, Grady Booch, Benjamin/Cummings, 1987
- (2) Software Engineering with Ada, Second Edition, Grady Booch, Benjamin/Cummings, 1987
- (3) Ada Programming Language, Military Standard ANSI/MIL-STD-1815A, 22 January 1983.

Michael I. Schwartz

EDUCATION:

BS, Mathematics, Case Western Reserve University, Cleveland, Ohio 1976

MS, Mathematics, Michigan State University, East Lansing, Michigan 1981

EXPERIENCE:

Michael Schwartz is a Staff Engineer currently involved in Ada research. His current direction is using Ada for designing and implementing operating system dependent utilities. Mr. Schwartz has been supporting research and development activities for over 7 years at Martin Marietta, including areas of status/readiness reporting systems, problems involving multi-level computer security and formal proofs of program correctness, software tools, and database methodologies.

Mr Schwartz has had extensive experience with the C, Ada, and FORTRAN languages, and some experience in GYPSY and LISP.

Randall Wm. Hay

EDUCATION:

BS, Mathematics, North Georgia College, Dahlonega, Georgia 1975

BS, CMS, Metropolitan State College, Denver, Colorado 1981

EXPERIENCE:

Randy Hay is a Software Engineer with Martin Marietta Corporation, I&CS Company, doing Ada productivity research. He has extensive experience with Ada, Model 204, and Cobol languages and has worked with FORTRAN, INFOBASIC, and Table Software.

*The opinions expressed in this paper are those of the authors, and not necessarily the opinions of Martin Marietta Corporation.

IMPLEMENTATION OF A REAL-TIME ELEVATOR CONTROL SIMULATION SYSTEM USING THE ADA LANGUAGE

David Bagley Kent Land Harold Tamburro Matthew Vega

**U. S. Army Communications Electronics Command
Center for Software Engineering
Fort Monmouth, New Jersey**

ABSTRACT

This paper describes an experiment in the development of software for a real-time system and graphics simulator, written in Ada. Specification of the problem was accomplished using the Hatley/Pirbhai (H/P) method.⁽¹⁾ The program was developed for use on Sun Workstations[®], and the graphics simulator takes advantage of SunCore[™] package of I/O primitives and interactive graphics.⁽²⁾ The experiment was conducted at the Center for Software Engineering, Fort Monmouth, NJ, to be used as a future test bed for software development methodologies.

1.0 INTRODUCTION

"The objectives of this experiment were to gain software engineering experience, use the Hatley/Pirbhai (H/P) methodology to specify a real-time software system, develop the system, and build a graphics simulator to represent an operational view of the system. Furthermore, a system had to be chosen that could be implemented in approximately six months time, demonstrate real-time behavior, incorporate a non-automated H/P approach, and graphically represent a system familiar to most people. With the above constraints, a multi-elevator controller and scheduler was selected for the experiment. Elevator scheduling was a problem presented at the 1986 ACM-sponsored workshop, on software development methodologies. Follow-up work was performed by Crawford⁽³⁾, using the PAMELA (Process Abstraction Method for Embedded Large Applications)^{TM(4)} methodology and automated tool.

SunWorkstation is a registered trademark of Sun Microsystems, Inc.

SunCore is a trademark of Sun Microsystems, Inc.)

PAMELA is a trademark of George W. Cherry

2.0 DEVELOPMENT BACKGROUND

The four members of the development team were interns in the Army Materiel Command's, Software Engineering program. The program consists of one year of training at the School of Engineering and Logistics, Texarkana, TX, and a second year of training at the Center for Software Engineering, Fort Monmouth, NJ. This experiment was performed during the second year of the Software Engineering program. Members of the group had previously not been involved in the development of a real-time Ada software system, nor had they worked together as a group.

3.0 SYSTEM OVERVIEW

Elevators contained in the simulation have many real world features. Some of these include summons buttons (up and down) on floors, destination buttons in elevators, direction and location lights on elevators, and elevator approach lights above elevators. Figure 1 illustrates the view seen during the elevator simulation. Not shown in this figure, but appearing dynamically during the simulation, are doors opening and closing, buttons/switches shading and clearing, and movement of people (passengers) in and out of elevators. During the simulation, people will enter elevators from the left side, and exit elevators from the right.

Additional features were added to simulate a real elevator. Such features included a run/stop emergency switch, an alarm button, an open/close door button, and a power on/off switch (for maintenance purposes). The run/stop emergency switch would stop the elevator in between floors and sound an alarm, if switched to the stop position while traveling. Otherwise, if it is switched to the stop position, while stopped at a floor, it would open the doors, keep them open, and sound an alarm. The open button stops the elevator door from closing. It, however, has no effect when the elevator is in motion, or if the door is already opened. The close door button closes elevator doors

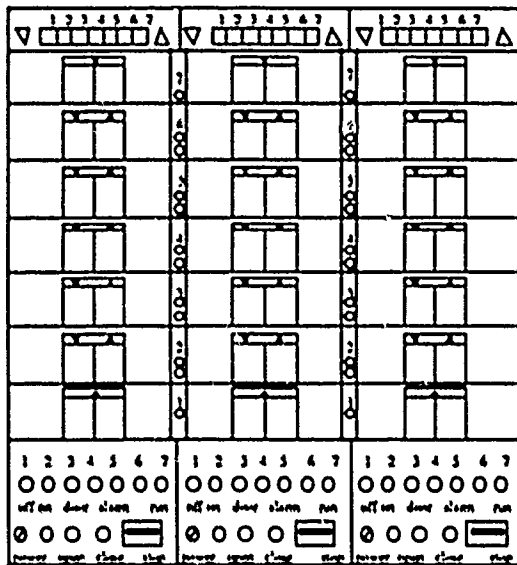


Figure 1.

faster than normal, by cancelling low priority levels associated with the close door function. Certain assumptions were made on the power on/off switch, since this switch is key operated on real elevators, and no real experimentation could be performed. The on/off switch was made to override all other buttons and switches for the elevator.

The elevator controller was designed to reflect a user's view of elevator systems. Components that are not seen by elevator passengers (ie., floor sensors indicating an approaching elevator) were not incorporated into the elevator controller program. The graphics simulator is a stand-alone program, originally written in the C language, then converted to Ada (with the exception of the SunCore dependent portions of C code). The elevator controller program is interfaced to the graphics simulator program, to form the complete elevator simulation system.

4.0 HATLEY/PIRBHAI METHOD

The I/P method is an extension of DeMarco's Structured Analysis (SA).⁽⁵⁾ The method extends SA to consider control flow, state machine behavior, and timing considerations in the requirements specification phase of development. Along with Data Flow Diagrams (DFD's), Mini-Specs (called PSPECS in I/P notation), and Data Dictionary (DD), comes Control Flow Diagrams (CFD's), Control Specifications

(CSPECS), and Timing Specifications. CFD's map control flow, just as DFD's map data flow. In addition, CFD's include a symbol (a slanted line) indicating an interface with a CSPEC. CSPECS specify the finite state machine behavior of the system. This behavior can be represented as a state diagram, process activation table, or matrix. CSPECS model real-time system activity, according to the I/P specification method. Timing Specifications are the allowable response time for the system to respond to a system input. Timing Specifications were not included in this simulation, except that the system had to respond to the user within a reasonable time limit. The I/P method consists of two models, a Requirements Model (RM), currently being discussed, and an Architecture Model (AM), which states the system's design structure. The development group did not construct an AM. It was felt that enough understanding of the system was obtained by the RM, in this case. For a larger, or more complex system, a AM may have been necessary.

5.0 SYSTEM DESIGN

Using the I/P method, the elevator design was broken into three sections: Event Generator, for the arrival of people, Elevator Controller, to schedule elevator operations, and Graphics Simulator, to simulate the functions of the elevator. Figures 2 and 3 are illustrations of the Control Context Diagram (CCD) and level 0 DFD/CFD, respectively, for the elevator simulation system. DFD's and CFD's are usually separated, but

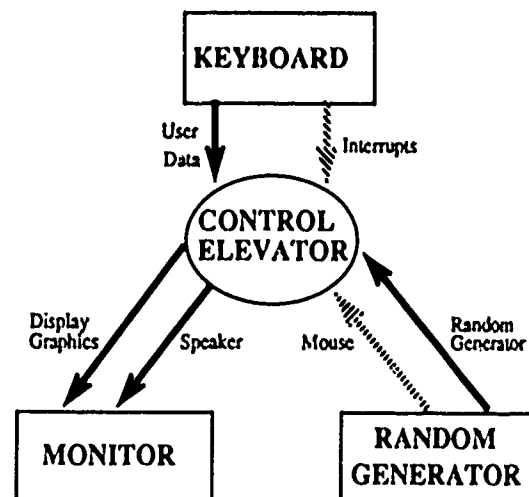


Figure 2.

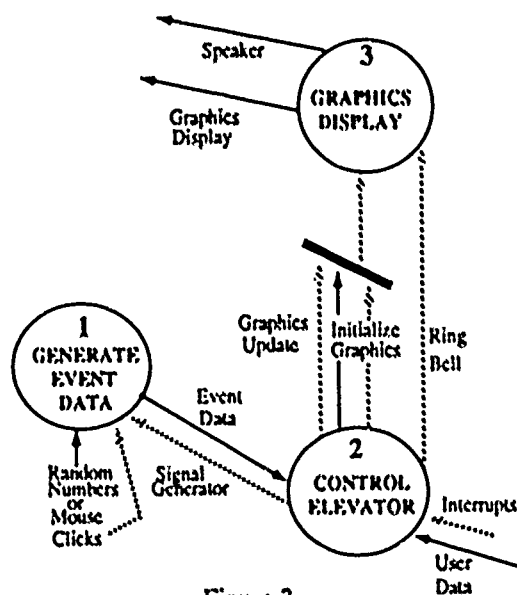


Figure 3.

were joined here, requiring less space.

In the CCD (Figure 2), the system is indicated by a circle, and the external entities the system communicates with are indicated by rectangles. Data flow is represented by solid lines, and control flow by dashed lines. In this case, the mouse was considered part of the Random Generator, and sends control signals to the elevator controller. The level 0 DFD/CFD (Figure 3), displays the three main functions of the elevator system. The slanted bar in the figure is the interface to a CSPEC, which will be described later.

5.1 EVENT GENERATOR

The Event Generator accounts for the arrival of people, and their service requests. This can happen by a random arrival generator, or by serial mouse input. The random arrival generator contains four sub-functions: an arrival generator, a starting floor generator, a destination floor generator, and a processing function. Throughout the execution of the simulation, the program checks to see if a summons was generated. When a summons is generated, a starting floor and a destination floor is then generated. If the destination floor is greater than the starting floor, an up call is produced. If the opposite is true, a down call is produced. If both starting and destination floors are equal, then there is no arrival. The random generator is designed to avoid inconsistencies, such as a passenger pressing

an up button, then pressing a destination floor below the originating floor. Finally, there is no random inputs to run/stop, open/close, and alarm buttons, or to the on/off switches.

5.1.1 SERIAL MOUSE INPUT

Mouse input allows considerable freedom in choosing a scenario for elevator service requests. The mouse can be positioned and clicked to select any button, or switch, on any elevator. Once a button is selected by the mouse, it cannot be undone. However, switches can be toggled on and off at will. Thus, the controller should be flexible enough to avoid being tricked by clever operators of the simulator. One potential problem is that more people can wind up leaving an elevator than had originally entered it. This may occur by clicking the mouse on more than one destination floor when elevator doors are closed. Thus, at every destination floor, at least one person would be seen leaving the elevator.

5.2 ELEVATOR CONTROLLER

The elevator controller is designed to direct the operation of elevators. The controller was first designed where each of the three elevators performed its entire task (arrival of passenger(s), movement of elevator, and departure of passenger(s)), one at a time, in a round robin manner. This initial approach was used for testing purposes, and was not intended to reflect a real world situation. Initially, the graphics, and the controller and random generator were developed independently. In addition, the functionality of the controller and random generator were tested separately.

The next major step in the development was to incorporate a tasking scheme. The program was designed to use $n+1$ tasks, where n is the number of elevators. Tasks were created dynamically for each elevator scheduler, and the additional task handles events, and the initiation of the other tasks. The initial design was revised to correct a significant problem. The problem concerned a race condition caused by multiple tasks competing for the same resource, the "graphics". SunCore graphics restricts applications to one open graphic segment at a time. A monitor routine was developed to prevent the application from opening more than one segment at any given time. This protection scheme was used to monitor the draw routines, fill routines, text routines, and for the initialize routine. The state diagram representing the control specification for initialize and update graphics functions (the thick black slanted line in Figure 3) is illustrated in fig-

ure 4. While the simulation is running, the system can assume one of thirteen possible states, before looping back to the simulation running state, and continue to loop in this manner throughout the length of the simulation. A cyclic executive model was used, and tasking (actually multiprogramming) required setting up time slicing. This was done by customizing the package

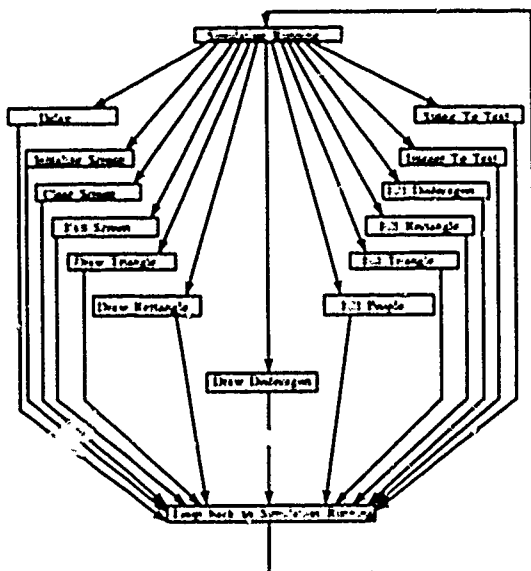


Figure 4.

Config, found in the VADS UNIX TM Implementation Reference Manual.⁽⁶⁾ The addition of adding tasking to the controller allowed multiple doors to graphically open and/or close simultaneously.

5.3 GRAPHICS SIMULATOR

The I/P method was not originally applied in the specification of the graphics simulator. An initial effort was established to test the capabilities of the SunCore graphics package. Members of the development team decided to spend several man hours on the development of a prototype graphics simulator, that ran through a predetermined format, to test every function required by the system. The graphics routines were originally written, from the bottom up, in C, as it was a language that supported a convenient interface to SunCore graphics. This was necessary because Ada does not have a direct interface to SunCore graphics.

UNIX is a trademark of AT&T Bell Laboratories

As the graphics simulator evolved, two generations of prototypes resulted before the top down design was completed. The first prototype was a static design, using three elevators and seven floors, while the second version was dynamic, allowing the user to select one or more elevators, and two or more floors. Both versions of the prototype were written in C. With lessons learned from the previous experience, a final prototype was developed that was more understandable, portable, maintainable, and reusable. This later version was developed and implemented primarily in Ada. Portability was improved by limiting the number of graphics routines. The future use of a different graphics package requires modifications to only a few routines. To improve maintainability, functions to change size and shapes of graphical entities was described mathematically. The graphics routines were designed and developed to be reusable for other systems requiring graphical simulation.

One of the problems encountered with the SunCore graphics package was that it contained no primitives to draw circles. One can easily write routines to draw circles, but filling a circle (to light and unlight a button) at a reasonable speed is a bit more complex. The best solution to the circle phenomenon was to use a dodecagon instead of a circle. Dodecagons can be easily drawn and filled on the user's screen, and for the purpose of a small button, it was a close approximation to a circle.

Draw, Fill, Text, and Initialize procedures are listed below, along with parameter list variables. These routines are not portable, because they depend on the graphics package being used (ie., SunCore). Since a monochrome screen was used during program development, color options were not implemented.

PROCEDURE	PARAMETER LIST
Draw.....	line thickness, location, size
Fill.....	shade, location, size
Text.....	integer/string, location, size
Initialize.....	none

Graphics procedures used in the Graphics simulator include: Draw Triangle, Draw Rectangle, Draw Dodecagon, Fill Triangle, Fill Rectangle, Fill Dodecagon, Fill People, Integer to Text, String to Text, Initialize, Clear, and Exit. Procedure Initialize sets up all parameters needed to draw on the user's screen. Clear and Exit functions are not used by the simulation, as the program was designed to run continuously.

Special considerations were involved in making the program as generic as possible. According to the requirements, three elevators and seven floors were specified. However, the program was designed to accept the number of elevators, number of floors, and starting locations of elevators, through user (mouse driven) or randomly generated input. The elevator graphical mode is designed to display a consistent, uniform graphical representation for any given number of elevators and floors.

6.0 CONCLUSIONS

The I/P method proved to be a useful approach for the design group. Use of the method resulted in a top-down, modular design, which specified real-time system requirements. The method was easy to follow, and lead from one step in the requirements specification process to the next, with no loss of understandability. Since an automated tool to implement this methodology was not available, the group felt a non-automated I/P method is best suited to small or medium size projects. This is not an indication of I/P's suitability to any particular size project, but rather a statement that large systems could produce hundreds, or thousands of diagrams, which would make the non-automated I/P method less practical to apply.

The importance of the use of some formal design approach is apparent. Which methodology is the "best" to apply to a given software project may be impossible to predict. What is important is that some formal methodology be followed and used (modified) according to the needs of the development team.

It is the intention of the elevator project group to design and develop several versions of the elevator controller software, each by a different software methodology. The controller's are to be used, in future experiments, with the existing graphics simulator. In order to help minimize some of the problems associated with interfacing hardware and software, an abstract interface should be included in future designs.

7.0 FURTHER INVESTIGATION

The elevator scheduler simulation was not designed to reflect a real world elevator system. There was no concept of a motor, nor were there floor sensors incorporated into the design. Actual movement of elevators is determined by buttons lighting and unlighting, and doors opening and closing. This is, after all, what an actual elevator user would see. Thus, the elevator simulation system was developed from a user

standpoint. In future experiments, the elevator design is scheduled to undergo a major modification from its current user standpoint, to an actual elevator designer standpoint (i.e., to reflect real world objects). The interface between the elevator controller and graphics simulator is to be changed so the controller portion of the program can be removed, and inserted into an actual running elevator system. Additionally, by using the existing graphics simulator as a test vehicle, and with continued redevelopment of the elevator controller, various software methodologies and CASE tools are expected to be evaluated. A version for the IBM PC and compatibles is currently under development, to demonstrate portability of the system.

ACKNOWLEDGMENT

The authors wish to express their thanks to Benjamin Casado, John LeBaron, Jay Seo, George Sumrall, and Tom Wheeler. The above mentioned contributed technical advice, direction, and support throughout the experiment.

REFERENCES

1. Hatley, D., Pirbhai, I., Strategies For Real-Time System Specification, Dorset House Publishing, NY, 1987.
2. "SunCore Reference Manual", Sun Microsystems, Inc., CA, 1986.
3. Crawford, B., "Building an Elevator Simulation in Ada Using a Process Oriented Methodology and Support Tool", Joint Ada Conference Proceedings, 1987, p. 223-230.
4. Cherry, G., "PAMELA2: An Ada-Based, Object-Oriented, 2167A-Compliant Design Method", Thought Tools, Inc., Reston, VA, 1988.
5. DeMarco, T., Structured Analysis and System Specification, Prentice-Hall, Englewood Cliffs, NJ, 1978.
6. "VERDIX Ada Development System Operation Manual for SUN UNIX", VERDIX Corporation, 1985.

ABOUT THE AUTHORS



David Bagley is a Software Engineer with the Center for Software Engineering, Advanced Software Technology, Fort Monmouth, N.J. He received his B.E. in Electrical Engineering from State University of New York at Stony Brook, and his M.S. in Software Engineering from Monmouth College, N.J. He is currently working in the Software Engineering Technology division, investigating Software Engineering Methodologies and Tool Assessment, and Requirements Engineering Technology.



Harold L. Tamburro is a Software Engineer with the Center for Software Engineering, Advanced Software Technology, Fort Monmouth, N.J. He received his B.S. in Chemical Engineering from the University of Pittsburgh, and his M.S. in Software Engineering from Monmouth College, N.J. He is currently working in the Software Engineering Technology division, investigating Software Engineering Methodologies and Tool Assessment, and Requirements Engineering Technology.

Kent Land is an Electronics Engineer with the Center for Signal Warfare, Warrington, VA. He received his B.S. in Mechanical Engineering from the University of Arizona and his M.S. in Software Engineering from Monmouth College, N.J. He is currently working in the Tactical Signal Intelligence Systems Division.

Matthew Vega is a Software Engineer with the MICOM Project Office at Redstone Arsenal, Huntsville, AL. He received his B.S. in Electrical Engineering from the University of New Orleans, and his M.S. in Software Engineering from Monmouth College, N.J.

PROCUREMENT OF AIR TRAFFIC CONTROL SOFTWARE IN Ada

Andrew C. Chung
Federal Aviation Administration Technical Center
Atlantic City International Airport, New Jersey

Summary

The Federal Aviation Administration (FAA) Advanced Automation System (AAS) prime contractor has selected Ada as the single High Order Language for the AAS. During the Design Competition Phase, the FAA requested the contractors to develop and carry out their Ada risk management plans, to complete their software top level designs in a compilable Ada-based Program Design Language, to conduct incremental detailed design walkthroughs, and to demonstrate their Ada readiness with Ada compiler benchmarks, Ada Programming Support Environment (APSE) tool demonstrations, and Software Engineering Exercises. During the Acquisition Phase, the FAA implements the AAS in three transition states with different Ada risk control goals: State 1, complete the development of essential APSE tools and automate en route Air Traffic Control (ATC) operations by introducing sector suites; State 2, modernize basic ATC data processing equipment and automate some terminal and tower ATC operations; and State 3, enhance ATC automation processing capabilities and automate the remaining terminal and tower ATC operations. The FAA encourages use of Commercially Available Software to minimize development and to facilitate future technology insertion.

Introduction

The Advanced Automation System (AAS) being developed at the Federal Aviation Administration (FAA) will modernize the United States Air Traffic Control (ATC) system. Ada has been selected as the single High Order Language (HOL) for the AAS. In this paper we will describe why Ada was chosen and how Ada risks are managed.

Background

The FAA en route air traffic controllers use 20 Air Route Traffic Control Centers (ARTCCs) to control all en route traffic in the continental United States. At present each ARTCC has a pair of IBM 3083 processors, one processor being in operational mode, and the other on standby. These 3083 processors, called the Host Computer, were deployed from late 1986 through 1988 to replace the previous en route processor, IBM 9020. During the rehosting, the en route National Airspace System (NAS) software underwent minimum changes. In addition to the en route facilities, the FAA also has approximately 400 Airport Traffic Control

Towers (ATCTs) for directing takeoffs and landings, and almost 200 Terminal Radar Approach Controls (TRACONS) for directing the movement of traffic at and in the vicinity of airports. The TRACON is often located one floor below the ATCT.¹

The AAS will introduce new workstations for en route, tower, and terminal air traffic controllers. It will also consolidate TRACONS and ARTCCs into new Area Control Facilities (ACFs), which will be located at the existing ARTCC locations. The AAS will be a distributed system; operations requiring centralized processing will be accomplished in the centralized computers, with all remaining functions performed within the individual sector suites.¹ A Local Communication Network (LCN) will perform all data transmission among these distributed processors. The overall Reliability, Maintainability, and Availability (RMA) design goal of the operational AAS segments is to continuously provide full service operation throughout their service life. Specifically, the full service mode of the final AAS state shall not break down for a period of more than 2.6 minutes per year. If a system fault ever occurs, the ATC services shall still be sustained or gracefully degraded. Therefore, layers of fault tolerance mechanisms, such as hardware redundancy, on-line system performance monitoring, automatic service mode switching, and facility backup will be employed to maintain essential ATC functions. Rapid isolation and replacement of faulty hardware parts and software components are also required to ensure fast recovery of full ATC services.

The AAS Design Competition Phase (DCP) contracts were awarded to teams led by Hughes Aircraft Company and International Business Machines Corporation (IBM) in August 1984. The DCP contract ended in July 1988, and the IBM team was awarded the AAS Acquisition Phase (AP) contract.

Selection Of Ada

The FAA required the AAS DCP contractors to select a single HOL for developing AAS software. Both contractors produced a "Language Selection Analysis Report" 9 months after DCP contract award. This report required that the contractors first identify all the HOL candidates which would be suitable for the AAS requirements, then detail the advantages and disadvantages of each according to specific language quality factors, and finally select one language as the HOL for the AAS.

The HOL candidates included these languages: JOVIAL J73, FORTRAN, C, Pascal, and Ada. Ada was selected over the other HOLs by both contractors. The rationale for their choice included factors, such as the better capabilities of the Ada language and the fact that Ada code is considered the best in areas of reliability, maintainability, and extensibility.

Reliability.

The major Ada features which contribute to a reliable software are strong typing, exception handling, and numeric precision. Strong typing helps maintain data integrity. Exception handling helps software fault detection and correction. Numeric precision ensures that a program can be expected to perform its intended function with required precision.

Maintainability And Extensibility.

The AAS must be maintainable not only to achieve the high system availability but also to reduce the life cycle cost. The incremental evolutionary development of the AAS requires software extensibility. Packages and generics are the major Ada features which contribute to its maintainability and extensibility.

Language Capability.

The AAS software will perform a wide range of functions including run-time system, information display, computation-intensive functions, data management, command and control, and message transmission. Ada has the potential to be able to do all of these functions.

Ada Risk Management In AAS DCP

Having reviewed the Language Selection Analysis Reports, the FAA was concerned about the immaturity of Ada technology and requested each contractor to develop Ada risk reduction plan and to report on their Ada risk reduction activities monthly. In late 1986, an FAA software team conducted a worldwide Ada usage survey, while a committee composed of Ada and software engineering experts was commissioned by the FAA to do an independent study on the use of Ada for the AAS.

Ada Usage Survey.

Alice Wong, who was the FAA AAS Software Technical Specialty Team Leader, led the Ada usage survey. The purpose of the survey was to gather information on Ada software development experiences and Ada risk management approaches. We interviewed about 24 Ada project personnel by telephone and also visited NASA Johnson Space Center at Houston, Texas and the Aviation Group of Transport Canada at Ottawa, Canada. Our major findings were reported to the AAS Program Director in January 1987, as follows:

Ada Compiler And Its Performance Were Essential To Project Success. As of November 1986, there were only 64 validated Ada compilers.

Performance of Ada compilers must be evaluated for specific applications. Shortage of good Ada compilers had resulted in waiver requests by some defense system implementors. Four projects surveyed had interim deliverables in Pascal. One project used an Ada Program Design Language (PDL) for designing the software but implemented it in "C."

Ada-Based PDL Should Be Used For Software Design. Twenty-three of the 24 projects surveyed had used or were planning to use an Ada-based PDL because it promotes the use of modern software engineering principles. A project could benefit from the use of Ada PDL even if the implementation language is not Ada. Most projects have customized Ada PDL guidelines.

Major Ada Projects Had Ada Programming Support Environment (APSE) Tailored For Their Specific Applications.

1. The Ada Language System/Naval (ALS/N) would provide program generation and execution support for mission-critical software targeted to standard Navy embedded computers. The primary focus was run-time performance.

2. The Worldwide Military Command and Control System (WWMCCS) Information System (WIS) had Software Development and Maintenance Environment (SDME).

3. National Aeronautics and Space Administration (NASA) Space Station Program (SSP) had a Request for Proposal (RFP) for the Software Support Environment (SSE) in 1986.

Ada Training Was Crucial For Ada Software Development. Good Ada programmers needed at least 6 months of training; 2 to 4 weeks of basic training and the remaining period spent in on-the-job training. Projects that had tried to shortcut formal classroom training had paid for it somewhere in the development process.

"Use Of Ada For AAS" Study Results.

The "Use of Ada for AAS" Study Committee, chaired by Dr. Victor R. Basili of the University of Maryland, recommended use of Ada, as well as risk reduction activities for the AAS.¹ To implement the committee recommendations, the FAA requested both DCP contractors to:

1. Complete their software top level designs for the Initial Sector Suite System (ISSS) and the ISSS System Support Computer Complex (SSCC-1) using an Ada-based PDL which can be compiled with a validated Ada compiler,

2. Conduct incremental Critical Design Review (CDR) software detailed design walkthroughs for the ISSS and SSCC-1, and

3. Demonstrate their Ada readiness with Ada compiler benchmarks, Software Engineering Exercises and APSE demonstrations.²

Ada Risk Management In AAS AP

The AAS will be implemented in three major states:^{3,4} (1) ISSS, (2) Tower Control Computer Complex (TCCC) and Terminal Advanced Automation System (TAAS), and (3) Area Control Computer Complex (ACCC). The AAS site transition spreads over a period of 6 years (August 1993, to November 1999). The transition plan and the long test and evaluation period of about 3 years for each state help reduce Ada risks. Lessons learned in earlier states will benefit later ones.

First State - ISSS.

The ISSS introduces the sector suite with display and input devices for en route controllers. The sector suite consists of one to four common controller workstations called common consoles, which will be used for all en route, terminal, and tower operations. The current Host Computer System (HCS) uses Plan View Displays (PVDs) for radar data display and flight strip printers. The common consoles which can display both radar and flight plan data will replace both the PVDs and flight strip printers. The current NAS software in the HCS will continue to perform the bulk of the ATC data processing, and send the processed radar and flight plan data to the sector suites for display. The emphasis of this state is the sector suite Computer-Human Interface (CHI).

The SSCC-1, which will be delivered with the first ISSS to the FAA Technical Center, will perform the functions of System Modification, System Testing and Verification, and Field Support for the ISSS. The Job Shop of the SSCC-1 will contain the APSE for developing, modifying, and maintaining the AAS Ada software. The SSCC-1 also has a Facility Configuration Console (FCC), a Stand-Alone Simulator (SAS), an ISSS System Support Facility (SSF), and common console simulators. For system testing, the SSCC-1 can emulate any deployed ISSS site.

The Ada risk reduction tasks for the first state are as follows:

1. Develop a complete set of APSE tools applicable for the AAS. The Ada compiler must be able to compile at least 1,000 executable source code statements per minute. The compiler library manager must be able to handle a large AAS-like software system containing more than 1 million source lines of code. The compiler-generated machine code and the run-time system for the common console processor must be sufficient to meet the sector suite workload, response time, accuracy, and RMA requirements.

2. Stress the importance of factory testing of Computer Software Units (CSUs), Computer Software Components (CSCs), and partial software builds. The software detailed design for ISSS and SSCC-1 was completed in the DCP.

3. Complete the ISSS acceptance test based on 18 pre-production units of the common console before authorizing an ISSS limited production release.

4. Complete the ISSS Operational Test and Evaluation (OT&E) based on the limited production model of the common console before authorizing the ISSS full production release. The total number of common consoles to be installed in 20 ISSS sites will be about 2,530.

Second State - TCCC And TAAS.

In the second state, the TRACON function of selected Automated Radar Terminal System (ARTS) facilities will be transferred to ARTCCs to form TAASs, while some of the corresponding ATCTs will receive TCCCs. The TAAS to be collocated with the ISSS will introduce sector suites to approach and departure ATC operations. The TAAS will receive flight data from the HCS in the same way as the ARTS III now does. Central processors will be used at the TAAS to perform the basic ATC functions which include Surveillance Data Processing, Automatic Tracking, Flight Plan Processing, Separation Assurance, Weather Processing, Digital Bright Radar Indicator Tower Equipment (D-BRITE) Support Processing, Ancillary Processing, Interim Altitude Processing, and Sector Suite Configuration and Sectorization.

The TCCC will introduce TCCC Position Consoles (TPCs) to the tower controllers. The TPC is comparable in nature to the common console of the ISSS and TAAS. According to the commonality requirement, both the TPC and the common console use identical processors and memory parts. Tower processors will be used at the TCCC to perform the basic ATC functions which include Surveillance Data Processing, Flight Plan Processing, Handoff of Controlled Tracks, Separation Assurance, Weather Processing, Airport Environmental Data Processing, ATC Mail, and Traffic Management.

The TCCC has two modes of operation; i.e., normal mode and stand-alone mode. In its normal mode, the TCCC utilizes the surveillance aircraft data and flight plan data obtained from the parent TAAS for display at the TPCs. In the stand-alone mode, when the communications between the TCCC and its parent TAAS are unavailable, the TCCC will perform limited surveillance processing and flight plan processing.

In this state, the SSCC-1 at the FAA Technical Center will have been upgraded twice, becoming first the SSCC-2 by receiving a TCCC SSF, and then the SSCC-3 by receiving a TAAS SSF.

The Ada risk reduction tasks for the second state are as follows:

1. Perfect the code generation and run-time system of the Ada compiler targeted to the central processor, tower processor, and TPC processor to meet the workload, response time, accuracy, and RMA

requirements of the TAAS and TCCC. The front end of the compiler is the same as that targeted to the common console processor. Therefore, the required compilation speed and capacity of the compiler will have been achieved during the first transition state. The AAS AP prime contractor has proposed to use a smaller model of the central processor for the function of Display Data Recording and Playback during the ISSS. Now the central processor will also be used for the TAAS basic ATC functions. The experiences in using the central processor during the first state will help reduce the risks in the second state.

2. Conduct incremental CDRs for ACCC and TCCC to complete their software detailed design in Ada PDL. The ACCC CDR includes the TAAS CDR as a subset. Because of the similarity of some basic ATC functions of TCCC and TAAS, reuse of their common source code is encouraged.

3. Stress the importance of factory testing of CSUs, CSCs and partial software builds.

4. Complete the FAA Technical Center acceptance test and OT&E of the TPC based on 24 units of the pre-production model before authorizing the production release of the TPC. The total number of TPCs to be deployed at 258 field TCCCs through the end of the third transition state is almost 1,450.

Third State - ACCC.

The ACCC is evolved from the ISSS sector suites and TAAS by adding more sector suites and ATC functions. The additional sector suites provide Traffic Management Positions, Oceanic Control Positions, and Flight Data Monitor Positions. The additional ATC functions are Automation Processing, Oceanic Processing, Facility Backup, Search and Rescue Data Extraction, Custom Trans-Border Detection Alert, Notice to Airmen Processing, and International Civil Aviation Organization (ICAO) Message Processing. The Automation Processing is the first implementation package of Automated En Route Air Traffic Control (AERA-1), which includes Flight Plan Conflict Probe, Sector Workload Analysis, Trial Flight Plan, Reconformance Aid, and Reminder Function. The Facility Backup capabilities allow adjacent ACFs to manage the airspace of an ACF that has had a catastrophic failure. Additional approach and departure ATC operations will be transferred from ARTS facilities to the ACCC, and the corresponding ATCTs will be equipped with TCCCs.

With the delivery of the first ACCC to the FAA Technical Center, the SSCC-3 will be upgraded to become SSCC-4 by receiving an ACCC SSF. Also in this state, a Research and Development Computer Complex (RDCC) will be created to contain a full

complement of ACCC and TCCC capabilities to support the testing of advanced ATC concepts, hardware upgrades, and new software functions.

The Ada risk reduction tasks for the third state are as follows:

1. Improve, if necessary, the Ada compiler targeted to the central processor to meet the workload, response time, accuracy, and RMA requirements of the ACCC.

2. Stress the importance of factory testing of CSUs, CSCs, and partial software builds.

The ACCC will have the capacity and functional capability to support fully integrated en route and terminal approach and departure ATC operations including future expansions. A planned future expansion is the AENA-2 which will extend the Automation Processing of the ACCC to provide the ATC controllers with Conflict Resolutions, Metering Actions, and Automatic Clearance Generation.

Use Of Commercially Available Software (CAS) For Advanced Automation System (AAS)

Since the FAA prefers use of standard commercial products to minimize development, the AAS contains many CAS items, such as compilers, loaders, LCN software, operating systems, screen graphic generator, and data base management system. The use of CAS not only reduces the amount of Ada software to be developed but also facilitates new technology insertion. For example, use of CAS operating system and Ada compiler allows us to adopt their future upgraded versions more readily. Most of the AAS CAS items are not Ada software because very few standard Ada CAS items are now available. However, when AAS software needs to be upgraded, we could replace some non-Ada CAS items with equivalent Ada CAS items, if then available.

Conclusion

The Advanced Automation System (AAS) contractor has selected Ada as the single High Order Language (HOL) to develop the AAS software. Sound software engineering practices are emphasized to design the Ada software. Ada Programming Support Environment (APSE) tools have been assembled to develop, test, and maintain the Ada code. An AAS transition plan which requires incremental delivery of Ada software in three states helps reduce risks. Use of Commercially Available Software is encouraged to reduce new software and to keep pace with future technology advances. The AAS development has benefited from the Ada technology accomplishments of the past. We continue to depend on the future Ada technology for successful AAS implementation and maintenance.

References

1. Basili, V.R., Boehm, B.W., Clapp, J.A., Gaumer, D., Holden, M., and Summers, J.K., "Use of Ada for FAA's Advanced Automation System (AAS)," MTR-87W77, The MITRE Corporation, McLean, Virginia, April 1987.

2. Montgomery, D.G., "The Use of a Software Engineering Exercise During Source Selection," The MITRE Corporation, FAA Technical Center, Atlantic City International Airport, New Jersey, to be presented at the Seventh Annual National Conference on Ada Technology, March 1989.

3. U.S. Department of Transportation, Federal Aviation Administration, "Advanced Automation System Transition Concepts and Requirements," FAA-TRO-AAP-003, 28 August 1987.

4. U.S. Department of Transportation, Federal Aviation Administration, "Advanced Automation System System Level Specification," FAA-ER-130-005H-AP, 28 August 1987.

Andrew C. Chung
FAA Technical Center, ACN-130
Atlantic City International Airport, NJ 08405



Andrew C. Chung is Computer Scientist in the Automation Division, FAA Technical Center. He has been a key member of the AAS software monitoring team, and is now leading the SSCC-1 development test and evaluation. Before joining the FAA in 1980, he had 7 years of experience in Computer Aided Design and Manufacturing (CADAM). He has a B.S. in Physics and an M.S. in Nuclear Physics both from Taiwan, and a Ph.D. in Geophysics from Massachusetts Institute of Technology. He is a member of ACM, SIGAda, IEEE, and Sigma Xi.

THE USE OF A SOFTWARE ENGINEERING EXERCISE DURING SOURCE SELECTION

David G. Montgomery

The MITRE Corporation

ABSTRACT

This paper presents a concept for the use of a Software Engineering Exercise (SEE) during the source selection process of a system acquisition. The development of acquisition-specific SEE requirements are discussed, along with the development of criteria to evaluate the offerors' performance of a SEE. Also addressed is the recommended composition of the evaluation team and the overall evaluation approach.

Finally, this paper concludes that the use of a SEE may not be appropriate for all source selections but is definitely an asset when used during large-scale system acquisitions.

1.0 INTRODUCTION

1.1 Background

The Federal Aviation Administration (FAA) has begun an extensive plan to both modernize and enhance the current Air Traffic Control (ATC) system. This plan, titled the National Airspace System (NAS) Plan[1], is currently composed of over ninety projects. The cornerstone of the NAS Plan is the Advanced Automation System (AAS). The AAS is designed to 1) replace aging equipment whose capacity and availability cannot meet the ATC needs of this decade and 2) replace the first-of-its-kind software, which is limited in extensibility, to meet the ATC functional and capacity needs through the end of the decade.

The development of the AAS project is broken down into two phases, the Design Competition Phase (DCP) and the Acquisition Phase (AP). The AP as specified by the FAA is analogous to the Department of Defense's Full-Scale Development Phase. Two contractors were selected to participate in the DCP. The main goal of the DCP was for each contractor to develop independently a design for the overall AAS that met all of the requirements contained in the FAA's System Level Specification.[2] Additionally, special emphasis was to be placed on the development of prototype hardware and software for the AAS man-machine interface (controller work station). At the conclusion of the DCP both contractors provided proposals for the actual development of the AAS project during the AP.

During the DCP, the FAA required that a single high-order language (HOL) be selected to implement the overall system requirements. Non-cost beneficial exceptions to this requirement were to be granted on a case-by-case basis only when fully justified by the DCP contractor and approved by the FAA. Anticipating that the contractor selected to implement the AAS might choose the Ada language as their single HOL and citing its lack of experience with Ada, the FAA commissioned a study to address the use of Ada for the AAS.[3,4] As part of this study, risk areas were identified and appropriate risk reduction activities were recommended. One of the recommended risk reduction activities to be conducted prior to the selection of the AP contractor was the conduct of a Software Engineering Exercise (SEE).

Since the information developed during the conduct of the FAA SEE was part of the source evaluation, it is considered source selective sensitive. Also, the SEE as specified as part of the DCP was unique to the AAS and not reflective of the general development of a SEE as depicted in this paper. Therefore, this paper will not cover the implementation of the FAA-specific SEE but, instead, will discuss SEE requirements in general and the steps involved in the conduct of a SEE. For the development of these general requirements, background information was taken from the exercises that have been published.[5,6,7,8]

1.2 SEE Description

A SEE is a small-scale system design exercise conducted by each offeror¹ and evaluated by the Government during proposal evaluation. It is intended mainly to evaluate the offeror's design methodologies as documented in both the Software Development Plan (SDP) and Software Standards and Procedures Manual (SSPM)² and demonstrated through the design of the exercise system. It also may be used to assess the offeror's software management, software tools, software code, and software testing techniques. This evaluation process is intended to allow the Government to assess the degree of risk associated with the offeror's software development methodology. Successful accomplishment of a SEE by

¹ The term offeror is used to refer to all respondents to the Request for Proposal.

² When referring to both the SDP and SSPM jointly, they will be called the "proposed plans" throughout the remainder of this paper.

an offeror will give some assurance, although no guarantee, that the offeror has the ability to complete the design phase successfully. On the other hand, the failure of an offeror to successfully complete a SEE provides some evidence that the offeror would have a low probability of success in completing the contracted tasks.[6] In addition, the results of a SEE, whether or not successfully completed, give some visibility into possible problem areas in the offeror's developmental methodologies and provide additional information to be used during the source selection process. This insight also identifies areas in the offeror's software engineering design methodology that must be given additional attention at the start of the acquisition phase.

Typically during a procurement, an offeror's proposed plans are evaluated during proposal evaluations. Areas of the proposed plans that are evaluated include the sections regarding the requirements analysis, design methodologies, and the coding techniques to be used.[5] Additionally, the proposed staffing levels are also reviewed. The evaluation of the plans is designed to give the Source Evaluation Board (SEB) some insight into the offeror's proposed methodologies concerning software development. This paper evaluation, however, does not provide any insight into the offeror's actual implementation of the plans. While they may be judged as adequate during the technical evaluation, the actual implementation of the plans may be well below the required SEB standards. As a result of improper implementation of the plans, schedule slippage and/or cost overruns are likely to occur. To help avoid an occurrence of this type, MITRE personnel developed a SEE as a non-standard method to be used during source selection for evaluating not only an offeror's SDP but also the offeror's expertise in the proposed software development approach.[5,6,7]

A SEE provides insight into the offeror's software development approach by testing the offeror's proposed methodology.[5] This is demonstrated through the offeror's design and, when required, implementation of a small exercise system. This exercise system provides the opportunity to evaluate the offeror's ability to use modern software engineering principles, implement the proposed plans, and organize a team for a SEE that is knowledgeable in both the proposed development methodology and the selected implementation language. Unless the implementation language is specified in the contract, a SEE is designed as an exercise independent of language and development methodology. It is formulated so that each offeror can demonstrate the proposed design and implementation methodology to be used during the actual system development activity.

2.0 SEE IMPLEMENTATION

The actual implementation of a SEE is divided into three activity areas: 1) preparation, 2) conduct, and 3) evaluation. A milestone chart depicting the overall implementation is contained in Table 1.

TABLE 1 MILESTONES

<u>Time Period</u>	<u>Activity</u>
Prepare RFP	Develop SEE Requirements. Conduct Dry Run. Develop Documentation. Develop Specific Evaluation Criteria.
Release RFP	Finish Developing Specific Evaluation Criteria.
Accept Proposals	SEE Development by Offeror. Initial Evaluation. On-Site Review. Final Evaluation.
Award Contract	

2.1 SEE Preparation

The preparation of the SEE is concurrent with the preparation of the Request for Proposal (RFP) package. Preparation for a SEE is divided into three overlapping tasks: 1) development of requirements, 2) conduct of a dry run, and 3) development of documentation.

During this period clear and succinct exercise system requirements are developed by the contracting agency. To be meaningful the exercise system must, as a minimum, be relevant to the mission of the new system. It must also have the appropriate size and complexity to allow the design to be completed within the Government-imposed time constraints. Unless the competing contractors are already under contract to the Government (as in the case of the AAS), a SEE requirement cannot be part of the actual system requirements. Therefore, the SEE requirements are written in such a way as to reflect the critical functional requirements of the proposed system (i.e., if, as in the case of the AAS, the proposed system is to be an I/O intensive system, then the requirements would be designed to stress I/O operations that are comparable with the proposed system).

To verify that the exercise system requirements are indeed reflective of the overall system requirements, a dry run should be conducted by contracting agency personnel who will be involved with the evaluations. During this dry run the exercise system requirements can be modified to ensure that the exercise requirements that are presented to the offerors are both distinct and succinct. A set of ground rules that must be followed by each offeror when conducting the exercise are also developed at this time.[6] Some examples of the ground rules that should be specified are:

1. Developmental time frame.
2. The make-up of the offeror's SEE team.
3. Required hardware (if appropriate).
4. Detailed requirements (e.g., performance requirements).

The documentation is then developed to include the appropriate stipulations in the Request For Proposal (RFP) package. The Instructions for Proposal Preparation (IFPP) and the Section M Evaluation Criteria are modified to include this documentation.[5] The IFPP must be modified to include the preliminary SEE instructions which describe the areas to be evaluated and the overall scope of the exercise. The Section M evaluation criteria are modified to present to each offeror the overall ground rules for the exercise and discuss the general evaluation criteria. Finally, the exercise system requirements are documented in detailed instructions that are provided to each offeror after the receipt of the proposals.[6]

During this time period, the detailed evaluation criteria that are to be used by the SEE Evaluation Team (SET) (see Section 2.3) are developed. These evaluation criteria are used during both the initial (in-house) and on-site evaluations (see Section 2.3). These criteria are established to provide consistent guidelines for the evaluation of the submitted SEE material.[8] These guidelines are also used to place emphasis on the most critical aspects of the exercise.

2.2 SEE Conduct

Upon submission of the proposal each offeror is provided with a set of detailed instructions that includes the exercise system specification, the specific ground rules for conduct of the exercise, and the time allocation for the exercise.

During the conduct of a SEE each offeror develops a complete software architecture for the exercise system to include a requirements analysis, a preliminary design, and a detailed design. As part of this design phase each offeror must follow the proposed design analysis methodologies, as documented in the proposed plans, that were submitted with the technical proposal. Also, all proposed tools to be used during the actual system development activity should be exercised to the maximum extent possible during the development of the exercise system.

At the end of the development phase, the offeror submits all requirements analysis and design products (both textual and graphic) that are generated as part of the SEE. This documentation should also include all intermediate products developed. All changes to the proposed plans that were identified as part of the exercise activity must also be submitted. After acceptance of the offeror's SEE products, there should be no interaction between the SET and the offeror's personnel. Also, after the acceptance of the submittal, no updating of the material should be allowed. For ease of review, all products that are submitted should be presented in both hard copy form and in machine-readable format.

2.3 SEE Evaluation

The SEE Evaluation Team (SET) should consist of members of the cognizant Government agency, and a

number of additional technical advisors. The evaluation team members should have a background in overall software engineering principles and be knowledgeable in both the offeror's proposed design methodologies and the proposed implementation language. The SET should be divided into evaluation groups according to individual expertise.[6] Although responsible for specific areas during the evaluation, frequent group interaction should be encouraged.

The evaluation approach is divided into three phases: 1) the initial evaluation, 2) the on-site review, and 3) the final evaluation. The time specified for each phase is the approximate time necessary for the conduct of each review phase for the type of SEE that is depicted in this paper. The actual times for each evaluation phase will vary depending on the specifics (complexity) of the actual exercise.

Upon receipt of the offeror's SEE products, the initial evaluation is conducted using the offeror's proposed plans. This review, employing the previously generated evaluation criteria, is used to determine if the exercise was developed in accordance with the proposed plans and if the proposed design methodologies are adequate to develop the overall product. The end result of this evaluation is the identification of the strengths and weaknesses of the offeror's SEE products and design methodologies. In addition, questions to be asked during the on-site review are generated.[7] Although no specific time frame is set for this evaluation, experience has shown that this initial evaluation should take no longer than one week per offeror.[7,8]

The second phase of the evaluation is the on-site review. The purpose of the on-site review is to verify the information gathered during the initial review and obtain additional information necessary to complete the exercise evaluation.[7] This on-site review, conducted by the SET after its initial evaluation of SEE products, consists of a briefing given by the offeror and should highlight:

1. Software Development Folders (SDFs) for each unit designed.
2. Tools used to conduct the exercise.
3. Changes to the SDF.
4. Lessons learned.
5. Record of staff time expended.
6. Experience level of personnel.
7. Estimate of computer resource utilization.
8. Amount of Quality Assurance interaction.
9. SEE team member's level of expertise in the proposed language.

During this briefing the offeror should provide answers to questions concerning various aspects of the exercise system. This on-site evaluation should last for approximately one day per offeror.

The third phase of the evaluation is the final evaluation. This period should be used to update the initial evaluation using the information gathered during the in-house review. This period

is also used to develop the formal evaluation report that is submitted to the SEB. The final review should last for approximately one week.

3.0 CONCLUDING OBSERVATIONS

A SEE can be an extremely beneficial source selection evaluation technique. Requiring each offeror to develop SEE products allows the SEB to evaluate what the offeror can really do, as opposed to what the offeror says can be done. The early visibility into the offeror's software development methodology, in conjunction with the review of the proposed plans, provides discriminating source selection information. This information includes the offeror's ability to implement the proposed plans, the offeror's expertise in the proposed design methodology and tool set, and the offeror's experience with the proposed design language and, when required, the proposed implementation language. [7,8]

A SEE provides an excellent vehicle with which to identify early problems in the offeror's software development approach. The requirement to develop an actual system design provides visibility into the proposed design methodology and the capability to identify incomplete methodologies. By uncovering problems early, the Government and the contractor are able to concentrate on these problems at the start of the acquisition phase (full-scale development) rather than waiting until the actual development phase has been completed and the deficiencies become more costly to correct.

Conducting a SEE during the source selection process can be costly to both the Government and the offerors. On both sides, it requires an investment of people, time, and money. The conduct of a SEE may also add significant time to the source selection. Although on the surface a SEE appears to be very beneficial, more studies are needed as to the cost/benefits ratio to determine if it should be required for every project. While it is apparent that a SEE is of great benefit to large projects (e.g., AAS), the value of the use of a SEE during source selection for smaller projects must be determined on a case by case basis.

Finally, a SEE can only be worthwhile during source selection if the exercise system requirements are tailored to the individual project. This fine-tuning of the SEE requirements can only be done by personnel that are intimately aware of the needs of the particular project. The use of an off-the-shelf (generic) SEE will not provide the source selection information or information necessary for addressing problems early after the contractor has been selected.

LIST OF REFERENCES

1. U.S. Department of Transportation, Federal Aviation Administration, "National Airspace System Level 1 Design Document," NAS-DD-1000 B, May 1986.
2. U.S. Department of Transportation, Federal Aviation Administration, "Advanced Automation System, System Level Specification," FAA-ER-130-005, August 1987 (with revisions 25 May 1988).
3. Bazill, V., B. Boehm, J. Clapp, D. Gaumer, M. Holden, J. Summers, "Use of Ada for FAA's Advanced Automation System (AAS)," MTR-87W77, The MITRE Corporation, Washington, DC, April 1987.
4. Salwin, A., "Briefings on the Use of Ada for FAA's Advanced Automation System (AAS)," MTR-87W87, The MITRE Corporation, Washington, DC, April 1987.
5. Maciorowski, S., "CCPDS-R Software Engineering Exercise (SEE): An Overview," WP-27213, The MITRE Corporation, Bedford, MA, March 1987.
6. Alex, G., G. Huff, P. Lasky, J. Maurer, S. Maciorowski, "CCPDS-R Software Engineering Exercise Experiences Volume I: Government Preparation and Dry Run," WP-27385, Vol. I, The MITRE Corporation, Bedford, MA, June 1987.
7. Alex, G., G. Huff, P. Lasky, J. Maurer, S. Maciorowski, "CCPDS-R Software Engineering Exercise Experiences Volume II: Use During Actual CCPDS-R FSD/P Source Selection," WP-27385, Vol. II, The MITRE Corporation, Bedford, MA, September 1987.
8. Howell, C., "AAS Software Engineering Exercise Assessment," The MITRE Corporation, Washington, DC, February 1987 (Unpublished).

ACKNOWLEDGEMENTS

This paper consolidates ideas developed by MITRE personnel in both Bedford and Washington and represents the work of many individuals. The author wishes to thank J. Gordon for her thorough review and helpful suggestions during the writing of the paper. The author also thanks M. Baden who typed the various drafts of this paper.

An Approach to Ada Compiler Acceptance Testing

E.G. Amoroso
T.D. Nguyen

AT&T Bell Laboratories

Abstract: This paper describes an ongoing acceptance testing approach being used at AT&T Bell Laboratories and AT&T Technologies to evaluate a port of the Verdix Ada Development System (VADS) to the AT&T 3B family of computers. The approach has been designed to exercise the compiler in as realistic a setting as possible from several different points of view. It is shown that the evaluation provides several other important benefits as well.

1. Introduction

During the past two years, AT&T and Verdix have been involved in an effort to port the Verdix Ada Development System (VADS) to the AT&T 3B family of computers. As the port nears completion, AT&T has begun to implement an acceptance testing approach that was designed to provide useful information about the *quality* of the VADS compiler and its environment. The approach is characterized by the following points:

- All components of compiler quality may be exercised effectively.
- The compiler is evaluated in a highly realistic setting.
- The compiler is evaluated from several points of view.
- Programmers with little or no background in Ada may contribute productively.
- Staff and resources may not have to be solely allocated.
- Existing and ongoing projects may directly benefit.
- The evaluation staff gains valuable Ada experience in realistic settings.
- The approach may also be used to "test" systems other than Ada compilers.

This paper provides a rationale for and description of the acceptance testing approach. A description of the implementation of this approach for the VADS compiler is also included.

2. Evaluation Alternatives

As stated above, as completion of the VADS compiler port nears, an investigation into the quality of the compiler in the target AT&T UNIX® System V environment has begun. The first step in the investigation was to examine potential acceptance testing alternatives.

The first approach considered involved simply using the Ada validation test suite to gain some experience with the compiler and its environment. It was thought that although Verdix had already run these tests, the process of re-running them at AT&T might provide valuable information about the quality of the compiler. The problem with this approach, however, (beyond the duplicated effort), was that many of the components of compiler quality could not be tested effectively. Running a collection of compilation tests that have already been developed is but a single measure of how well a compiler will help human beings get their jobs done. Other important components of quality such as *reliability* (producing identical results for identical inputs), *usability* (helping new users without hampering experienced ones), *code efficiency*, and *support*, must also be taken into account in any acceptance testing approach. In addition, such an approach would provide little information about how well the compiler fit into the target UNIX environment.

With this in mind, a second potential acceptance testing approach was considered based on the design of a collection of our own compiler *penetration* tests. Assuming there were no obvious bugs in the compiler, these penetration tests would most likely explore the dark recesses of the VADS compiler in search of obscure bugs. The development of these tests would obviously provide valuable information about the correctness of the compiler, as well as information on the usability, reliability, and other components of quality. The problem with the approach, however, was that it would have encouraged most of the effort to be placed in areas of the compiler that are rarely used (if ever). The bulk of one's attention ought to be placed on those areas that will be most used (i.e., a *robust* approach). In addition, good penetration tests require the full attention of a dedicated staff, and once the tests are completed, they have little value beyond their intended function. For these reasons, the "penetration" acceptance testing approach was rejected.

A third approach considered for the acceptance testing involved simply putting the compiler into immediate use on an existing Ada project. Obviously, such an approach would provide a highly realistic evaluation, since the compiler would be exercised in a real setting. However, the approach was rejected for two reasons. First of all, a particular project might require heavy reliance on one feature, but might at the same time not require another

feature at all. This situation could bias the evaluation somewhat and allow certain aspects of the compiler to go unevaluated. A second reason the approach was rejected was that it was not feasible for any ongoing AT&T Ada-related project to terminate the use of the existing Ada compiler without severely impacting its progress and the schedule.

Finally, a compiler acceptance testing approach was agreed upon that provided most of the benefits of the above mentioned alternatives without many of the drawbacks. In fact, this approach provided several benefits that no other alternative could provide. The approach is based on the notion that ongoing non-Ada projects and work efforts can be used as a framework for evaluating an Ada compiler. That is, work that is already being planned or started that does not carry an explicit "programming language" requirement, could be done in Ada using the compiler to be evaluated. This approach has several attractive side-effects. For one, it tends to provide a well-rounded set of realistic situations within which to evaluate the compiler. For instance, several in our local community at Bell Laboratories were interested in providing a secure distributed bulletin board capability. Although the original plan had been to build the system using C, there was no reason why it could not be done in Ada. A second important characteristic of the approach is that resources and staff may not have to be explicitly allocated. If the work is being planned anyway, the evaluation does not cause a significant allocation increase. Also, unlike the penetration tests, these "tests" will continue to be useful even after the evaluation is complete. Perhaps the most important aspect of the approach, however, is that it forces programmers and engineers who would not normally be involved in such an effort, to gain Ada development experience.

3. Applying the Approach

Two important considerations were taken into account in selecting the type of work to be done in Ada for the evaluation:

- That several concerns (e.g., security, networking, performance), were likely to be of the utmost importance in future applications that might use the compiler.
- That the work should be of varying degrees of difficulty, and should cause as much of the compiler to be exercised as possible.

In this section, three ongoing AT&T efforts are described that, collectively, seem to meet the above considerations. It should be mentioned, however, that this work is ongoing, and in fact, more efforts may be added in the

future. It should also be noted that the very nature of the approach is quite conducive to such additional work efforts. One final point worth mentioning is that the approach is also appropriate for evaluating a compiler before it is purchased and ported.

3.1 A Novel Password Generator

Computer security has become an important concern in most of the areas that Ada is likely to be used (e.g., critical embedded applications, communications systems, etc.). With this in mind, it was determined that our acceptance testing should include some security-related work.

It turned out that the local systems engineering team involved in the design and development of *System V/MLS*, a security-enhanced UNIX System V-based operating system [Flink 88], had been interested in prototyping a rather novel approach to password generation. The approach combines the best features of automatic password generation (i.e., the removal of all blatantly bad passwords from a system), with the best features of user-defined passwords (i.e., their mnemonic nature).¹

In a system that supplies an automatic password generator, a fixed length password is generated, and although users are allowed several choices of strings, eventually one of the strings has to be chosen. Because the user has no control over its contents, the password is rarely mnemonic. (It is worth mentioning that pronounceable does not imply mnemonic!)

In systems that allow users to generate their own passwords, on the other hand, the mnemonic issue is often taken to its extreme. Rather than having a collection of passwords that are of an equivalent nature (in terms of their guessability), as in a system with an automatic password generator, systems with user defined passwords usually have some excellent passwords, some average ones, and some terrible ones. Combining this fact with the notion that an operating system is usually about as secure as its weakest password leads to a distressing conclusion about such systems.

The combination system that is being coded in Ada would allow a string of fixed length determined by an administrator, to be passed to users. Users would then "change" this string to a password that is both mnemonic and pronounceable. The rules for such "changing" could allow prepending, inpending, or postpending. For

1. The approach was first described to us by the AT&T Bell Laboratories Computer Center support group.

example, if the string 'U'd' was passed to a user by the password generator, then the user could change the string to 'UNIX'n'Ada' thus preserving the ordering of the original string.

Obviously, our description of this system is not nearly complete and only provides a 'flavor of what is being built, but the description points out several issues that are relevant to our present discussion. First of all, writing an Ada program to implement such a feature is not an unusually difficult exercise. As a result, this is a good example of an effort that is suitable for a novice programmer to tackle. By using a novice programmer in the acceptance testing, one stands to gain valuable information about the usability of the compiler from a novice point of view. In addition, valuable feedback on error messages, user documentation, and environmental tools is also obtained. A second important issue is that the program requires use of most of the "mainstream" features of the Ada language - the so-called Pascal subset. Such features are often used in Ada programs, and are thus most appropriate for acceptance testing. A final issue is that the program will clearly remain usable long after the acceptance testing has been forgotten. This valuable side-effect of our approach cannot be underestimated.

3.2 C++ vs. Ada Benchmarking

The issue of performance is one that clearly needed to be addressed as part of the acceptance testing. Ada performance is a concern that had been coming up repeatedly during the preparation of language trade studies for several Ada-related proposals, and many of our colleagues had expressed a particular interest in comparing the performance of Ada to that of the object-oriented language C++ [Stroustrup 86]. As a result it was decided that as part of the acceptance testing, a collection of benchmark programs (some of which already existed in C++) would be coded in Ada and C++ to determine the relative code efficiencies. Obviously, such an effort would also reveal other important comparisons as well, such as the relative usability and reliability of the two languages.

An important characteristic of this benchmarking effort is that it does not require a specified level of minimal Ada expertise. The complexity and accuracy of the benchmarks would, of course, depend on the experience of the programmer, but clearly, even a novice Ada programmer could design useful benchmarks that would provide valuable information. Another desirable characteristic of this benchmarking effort is that it involves C++ programmers and enthusiasts in the Ada testing effort. Under more traditional acceptance testing circumstances, these programmers might not be interested in testing the Ada compiler, but given the challenge of

comparing it to a language that they are interested in, provides the necessary incentive.

It should also be noted that the results of the benchmark remain useful (e.g., for proposal trade studies, etc.) after the acceptance testing has been completed.

3.3 Distributed Bulletin Board

A third important area that is already having a major influence on many Ada-related projects and efforts is secure networking. As a result, it was determined that the acceptance testing effort should include some secure network-oriented project. It turned out that several in our community at AT&T Bell Laboratories had been interested in providing a local secure distributed bulletin board system across our distributed configuration of secure computers running System V/MLS. Although initial prototypes of the system had been built in C, there appeared to be no reason why it could not be built in Ada.

The bulletin board would maintain a record of all privileges associated with a particular user. Reports posted to the bulletin board would be readable by only those users with the appropriate privileges. Reports posted to the board would inherit the privileges of the user posting the report. An administrator would maintain the contents of the board.

Implementing such a system in a distributed configuration of secure computers would likely require the use of most of the features of the Ada language including packages, tasking, generics, and exceptions. It would also require such features as the interface to the C programming language. As a result, such a project would require at least some Ada programming maturity. The project would result in feedback on the quality of the compiler from a more experienced user. The usability, reliability, documentation, environment, and code quality of the compiler would all be re-evaluated, but from a more sophisticated point-of-view, and in terms of *systems*, rather than just programs.

4. Concluding Remarks

In conclusion, an acceptance testing approach has been described that combines the best features of several traditional acceptance testing approaches into a collection of small projects that result in work that is useful even after the acceptance testing has been completed. It should be pointed out that although the approach has been described in terms of the AT&T/VADS porting effort, the approach is not Ada-specific at all, and could be used for the testing of any system that is being considered for installation or use.

5. Acknowledgements

The authors wish to thank Angie Brame and Tina Gulley of AT&T Technologies and for their valuable assistance in this effort. Neal Oliver provided several useful comments on the paper.

6. References

[DoD 83] *Reference Manual for the Ada Programming*

Language, United States Department of Defense, January, 1983.

[Flink 88] Flink, C.W. and Weiss, J.D., System V/MLS: Mandatory Policy Alternatives, *AT&T Technical Journal*, 1988.

[Stroustrup 86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.

SOFTWARE METRICS ANALYSIS OF THE ADA REPOSITORY

Ronald J. Leach

Department of Systems & Computer Science
School of Engineering
Howard University
Washington, D.C., 20059

1. INTRODUCTION

There has been a considerable amount of interest in the use of software metrics to describe the degree to which a piece of software possesses a given attribute. Metrics have been developed using the idea that the primary complexity of a program is the number and type of operands and operators, or the number of program branches, or the degree and type of interconnection between the modules that make up the program.

This work has assumed special significance in the Ada environment. Ada was designed in large part to support the development of large software projects which are built using the techniques of data abstraction. Thus an important use of metrics is in assessing the quality of Ada software projects. It has been clear for some time that no single number suffices to measure the quality of a piece of software. Typically, a piece of software has many attributes (portability, readability, verifiability, complexity of underlying algorithm, complexity of implementation, etc.) which are measured by various software metrics. Much of the research on software metrics is difficult to replicate because of the proprietary nature of the code being measured.

The references [1], [5], and [9], describe some general purpose results in this area, while the papers [3], [4], [6], [7] and [8] describe some metrics that are specific to the language Ada.

In this note, we explore some of the attributes of a large block of code written by a large collection of programmers with the specific intention of making the code available for common use. The code considered here is the Ada Repository, which at the time this paper was written (July 18, 1988) contained more than 54 megabytes of text with approximately 1.1 million lines of code and the rest documentation. Much of the emphasis of the repository is on the portability and reusability of the code, since

this was a primary reason for the creation of the repository. The other major emphasis is on the maintainability of the code. Code can be ported effectively from one installation to another only if the purpose of the code and any specific requirements about related hardware and software are understood.

We note that there is little discussion of the theoretical nature of either metrics in general or Ada specific metrics in particular in this paper; the emphasis is on determining essential attributes of precisely what currently exists in the Ada Repository. We consider the Ada Repository as a testbed for research in software metrics and note that the availability of the code to other researchers can aid in the development of software quality metrics.

This work is part of a research project at Howard University to apply metrics driven design to large software systems during most phases of the software life cycle. This project (called HUMAN for Howard University Metrics ANalyzer) currently consists of software tools for the analysis of programs written in C and Ada, with the tools being written in the appropriate language. The work described here involves the Ada version of HUMAN which is written in the language Ada itself. The Ada code for HUMAN is written in Meridian Ada running on AT&T PC6310s, which are IBM PC AT class machines. (The repository itself contains several metrics files in the directory METRICS. However, the McCabe cyclomatic complexity package requires access to the source code of an Ada compiler and the Halstead Software Science Metric package contains over 1.1 megabyte; this is far too large for an AT class machine. Therefore we wrote our own code. The source code is quite small and the Meridian compiler generates fast code for the 80286 processor.)

2. THE ADA REPOSITORY

The Ada Repository is a collection of packages which is made available to users for the cost of the distribution medium. The Ada Repository is stored on the Defense Data Network and is also available from other

sources. The Ada Information Office should be contacted for information on accessing the Ada Repository. The original intention of the Ada Repository was to provide Ada users and researchers with an opportunity to use a large variety of Ada packages in their work. One major goal of the Repository was to provide a set of software components which can be used in a variety of applications. Benefits include the dissemination of information to many people about the facilities available in the language Ada. Much of this information is textual and does not contain Ada source code. For example, the directories ANSI-LRM, CROSS-REFERENCE, EDUCATION, ID-FILES, NEWS, and WIS-ADA-TOOLS are approximately 4644000, 119000, 1670000, 1684000, 3534000, and 2450 bytes large respectively, and contain no Ada source code.

There are a total of 340 packages in the repository, written by more than 91 different programmers or programming teams. Thus the repository represents a cross section of the nature of existing code. The mean number of comment lines in the repository is 22.6%. Since only 23.7% of the files in the repository are source files, only about 20% of the repository consists of lines of Ada source code. The repository represents a cross-section of the nature of existing code. Many people have commented that the code in the repository is of uneven quality; we see this as an advantage in determining the actual state of affairs in the Ada community. It is not reasonable to only consider the quality of code written by outstanding programmers; in fact only 50% of Ada programmers are above average as Ada programmers.

There are frequent additions to and deletions from the Ada Repository. Thus to have a common basis for evaluation, we consider the Ada Repository as it existed on June 1, 1988.

The Ada Repository consists of 1433 Ada source code files, documentation, and test files. It also contains the <ANSI-LRM> (ANSI Language Reference Manual), <ONLINE-DOC> (on-line documentation), and <ADA> (information on how to use the Ada Repository) directories. In this paper, we only discuss those directories that contain source code. The files are grouped in the following directories that are indicated in table 1.

Of the 340 Ada source code files in the repository, data is presented here on 270. The 70 omitted files were not considered because they contained more than 6000 lines of code each. Our experience with the HUMAN metrics analysis program is that an Ada source code program of 6000 lines of code generates at least 20000 separate tokens and this exhausts the memory of the PC used for this project because of the length of the tokens and the need for associated data structures.

3. METHODOLOGY FOR METRICS COLLECTION

The Ada Repository is distributed in several formats. The format chosen for this project was UNIX tar format. The repository was loaded onto an AT&T 3B2/400 computer which at the time did not have an Ada compiler but did have a reliable tape drive. In order to read all of the files, the system variable *ulimit* had to be increased to

allow files of length greater than 1 MB. The files were downloaded to AT&T PC 6310's which are IBM PC AT class machines. All of the automated metrics analysis of the repository was done on the PC's using Meridian Ada. Because of the limited capacity of the PC's, it was necessary to split up a few files and to remove a few formatting commands included in certain files. The few files that were divided had all parts subjected to automatic metrics analysis and the totals for the files are presented in this paper.

The primary automatic metrics computed and presented in this paper are the Halstead Software Science Effort and the McCabe Cyclomatic Complexity. Halstead's metric is obtained by classifying all executable statements as being composed of operators such as $=$, $($, $)$, $*$, $+$, etc. and operands such as x , 7 , etc. The Software Science Effort is obtained by multiplying the expression

$$(N_1 + N_2) \log(N_1 + N_2)$$

by a constant called the Stroud number which is often estimated to be 18. Here N_1 , N_2 , N_1 and N_2 are the number of operators, operands, distinct operators, and distinct operands, respectively. This metric was designed to correlate with the number of "mental discriminations" performed by the programmer. Since we are interested in the correlation between attributes of the software and the values of various metrics applied to the code, we have ignored the Stroud number and used the natural logarithm in Halstead's metric. For simplicity, we have also rounded the value of the Halstead metric to the nearest integer.

McCabe's metric ignores assignment statements and instead considers the flow of the program. All statements in a program are considered vertices of a graph. Edges are drawn between vertices if there is direct connection between two statements by a loop, conditional branch, or the statements are in sequential order. McCabe's metric is

$$V - E + 2P$$

where E is the number of edges, V is the number of vertices and P is the number of separate parts (= number of subprograms called, including the main program).

McCabe [5] suggests that functions or procedures with a cyclomatic complexity greater than 10 should be avoided as being unstructured; the only exception to this rule of thumb would occur only when there are rare statements with many alternatives.

For each Ada source code file, we measured the values of the variables

PU = number of program units

HIT = total Halstead

MT = total McCabe

MII = Maximum Halstead of any unit

mII = minimum Halstead of any unit

MM = Maximum McCabe metric of any unit

mM = minimum McCabe metric of any unit

L = number of lines

common number of lines with comments

The results of the measurements are given in table 2.

The source code for the metrics analysis was approximately 3000 lines long and was 68200 bytes long (exclusive of documentation). The code was placed in several Ada source code files. This is considerably shorter than the combination of nearly 30,000 lines in the file *halsteadsrc* and the additional 10,000 lines in the file *mcabesrc*. The *halsteadsrc* file seems to contain several attempts at writing the Halstead metrics analysis so that a fairer comparison is between the 3000 lines of HUMAN and the approximately 20,000 lines for metrics analysis in the metrics directory.

4. ANALYSIS

The first question we considered was how the variables correlated with one another. The graphs in figure 1 suggest that they are related. However, the Halstead and McCabe metrics were so large for a relatively small number of files that the correlations were not very good. The correlations are given in table 3. The correlations are greatly affected by a few files having very large values of the Halstead or McCabe metrics, number of program units, number of lines, or the percentage of comments. A common statistical technique is to ignore these outliers when computing the correlation coefficients. We did not do that in this data set because we expect to expand the data collection to larger Ada source code files. The low correlations are likely to be increased when the additional large Ada source files are analyzed.

The correlation did not improve if we replaced a quantity with its density; that is if we replaced a variable such as *IT* by the quantity *IT/L*, the correlations showed little change and that change generally did not increase the correlation. These results are summarized in table 4.

The Halstead and McCabe metrics are called microlevel by some authors because they only consider the actual code within program units and do not consider the level of interconnection between the program units. In [6], the authors described computer programs for measuring the level of interconnection between Ada packages using the number of *WITH* statements in the packages and comparing this to the number of packages. No metrics for measuring the interconnection between program units were used in this paper since nearly all of the code in the repository is composed of files with one package per file. In addition, most of the code was not apparently intended as stand-alone code rather than for components in the development of large systems; most of the exceptions to this are in the *Components* directory. Consequently, no measurements of interconnections between program units are given in paper.

Portability and reusability issues are valid topics for research since the Ada Repository originally intended to showcase portable and reusable code. We begin our discussion of these issues with a survey of some existing work. In [4], several features of metrics used for measuring the maintainability, portability, and understandability of packages were discussed. These metrics included measurements of the following: *packages*, *generics*, *interfaces*

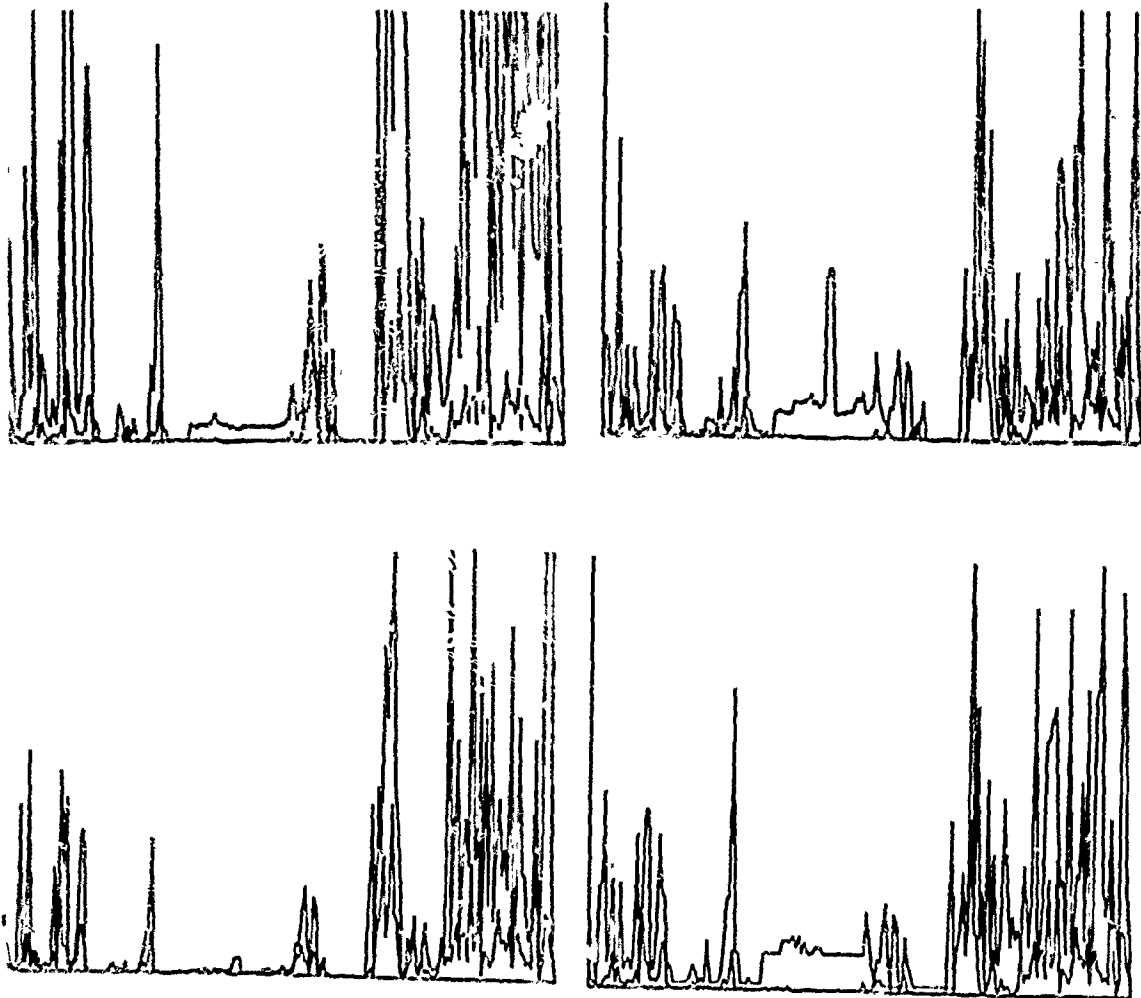
to other languages, and machine code insertions. Other factors in that paper considered were those due to verifiability, programmer understanding, and language level. It was shown in that paper that no metric applied at a fixed time in the life cycle could provide an accurate view of the run-time complexity of a program which permits the dynamic activation of tasks.

In general, these metrics had little validity for the code in the Ada Repository. The code had no machine code insertions and very few direct interfaces to other languages other than the language of the compiler in the *mcabesrc* file in the metrics subdirectory.

The directory *COMPONENTS* in the repository contains code intended to be used as software components in building reusable software. There are 43 Ada source code files in this directory which include 55 generic packages that are to be instantiated when used in other software. For example the file *stackada* contains a set of generic packages which may be used for typical stack operations such as pushing an object onto the stack, popping an object off of the stack, clearing the stack, or checking the stack for being empty.

The file *sortarrayada* contains generic sorting routines for which the formal generic subprogram parameter "*e*" must be specified with any instantiation; this is especially important for sorting records or access types. It contains a large sorting function which can call any of the common sort algorithms such as quicksort, bubble sort, *Insert* (which uses *B-trees*), selection sort, heap sort, insertion sort, and merge sort. Several of the sort algorithms have both recursive and non-recursive versions. All of the sort algorithms are appropriate for internal sorts only. The documentation suggests splitting files into manageable pieces, sorting the pieces, and combining them with a merge sort. This procedure might also be appropriate for external files.

A simple experiment was designed to determine some of the problems in portability and reusability of Ada packages. The source code file *filecomp.adb* was chosen as the file to be tested for portability. An attempt to compile the file showed that the compilation unit *TOD_UTILITIES* also needed to be compiled. The package *TOD_UTILITIES* was found in the file *todada*. This information could not be determined directly using the packages but had to be determined by either using the operating system or an external database. In addition, there are restrictions on the order in which packages can be compiled. This phenomenon in determining dependence and compilation order is well-known; it is the reason for many articles describing library maintenance and source code control and for the existence of the subdirectory *compilation-order* of the Ada Repository. The package *TOD_UTILITIES* did not resolve all of the difficulties; compilation of *TOD_UTILITIES* required compilation of another package called *SEARCH_UTILITIES* which was found in the file *searchada*. As before, identification of this compilation unit required some information external to the file. It is clear that this process can be quite lengthy in many situations.



DIRECTORY	FILES	ADA FILES
ADA-SQL	31	5
AI	15	1
ANSI-LRM	51	0
BENCHMARKS	41	13
CAIS	5	2
CAIS-TOOLS	3	1
COMPILATION-ORDER	12	1
COMPONENTS	94	42
CROSS-REFERENCE	5	1
DBMS	4	1
DDN	41	11
DEBUGGER	18	2
EDITORS	24	6
EDUCATION	41	0
EXTERNAL-TOOLS	13	4
FORMGEN	7	3
GENERAL	45	1
GKS	9	2
ID-FILES	175	0
MANAGEMENT-TOOLS	36	4
MASTER-INDEX	18	0
MATH	52	13
MENU	23	2
MESSAGE-HANDLING	13	2
METRICS	49	4
ONLINE-DOC	7	1
PAGER	10	4
POINTERS	35	0
PDL	28	3
PIWG	197	172
PRETTY-PRINTERS	23	3
SIMULATION	11	2
SPELLER	9	1
STARTER-KIT	6	1
STUBBER	9	2
STYLE	42	7
TOOLS	53	10
TRANSLATORS	45	15
VIRTERM	10	2
WIS-ADA-TOOLS	166	0

TABLE 1: DIRECTORIES

DIRECTORY	FILES USED	PU	IIT	MT	MII	mII	MM	mM	L
ADA-SQL	5	92	49991	220	5000	4	11	1	2307
AI	1	32	15125	96	824	17	14	1	1048
BENCHMARKS	7	6	0.740688	225	3334	33	23.7	1	2255
CAIS-TOOLS	1	66	33958	238	7249	11	47	1	4442
COMPONENTS	40	16.4	10316	40	1086	52	11	1	763
DBMS	1	66	55906	235	4259	29	63	1	2445
DDN	5	57.4	43316	276	3260	9	27	1	2375
EDITORS	3	80	42548	266	3917	6	40	1	2574
MANAGEMENT-TOOLS	1	5	12519	41	3848	871	18	1	579
MATH	11	27	25074	99.6	3581	29	17	1	1138
MENU	1	83	42330	200	2056	6	20	1	3973
ONLINE-DOC	1	43	24341	170	1199	14	19	1	2260
PAGER	4	22.5	13546	84.8	1126	90	13	1	958
PIWO	128	8.2	1429	14.5	615	362	3.5	1.7	133
PRETTY-PRINTERS	2	40	43106	187	1647	3	38	1	3876
STARTER-KIT	1	34	15152	74	992	4	6	1	1072
STUBBER	2	44.5	48084	231	7248	3	49	1	2428
STYLE	2	23.5	19591	122	3825	2.5	48	1	4084
TOOLS	6	50	23857	161	2157	193	15	2	1472
TRANSLATORS	3	63.3	136284	531	6559	12	43	1	3905
VIRTERM	1	25	5199	52	914	14	11	1	747

TABLE 2: METRICS PU = PROGRAM UNITS, IIT = Halstead TOTAL, MT = McCabe TOTAL, MII = Halstead MAX, mII = Halstead min, MM = McCabe MAX, mM = McCabe min, L = Number of Lines, comm = comments

	PU	IIT	MT	MII	mII	MM	mM	L	comm
PU	1.00	.575	.760	.369	-.192	.475	-.170	.644	-.232
IIT		1.00	.781	.692	-.131	.648	-.125	.705	-.117
MT			1.00	.567	-.157	.755	-.141	.755	-.129
MII				1.00	0.060	.703	-.060	.607	-.092
mII					1.00	-.102	.537	-.168	.013
MM						1.00	-.098	.677	.021
mM							1.00	-.163	.206
L								1.00	-.068
comm									1.00

TABLE 3: CORRELATIONS

DENSITY	PU	IT	MT	MTL	mll	MTM	mM	L	comm
Halstead per line	.230	.640	.309	.395	-.009	.304	-.034	.217	-.077
McCabe per line	.456	.006	.224	.009	-.107	.083	-.041	-.016	-.120
Halstead per program unit	.010	.519	.213	.569	.458	.348	.270	.213	-.062
McCabe per program unit	-.081	.215	.256	.438	.331	.422	.505	.146	.075
Lines per program unit	-.185	-.153	-.088	.322	.010	.223	.123	-.129	-.166

TABLE 4: MORE CORRELATIONS

	ITL	MTL	ITPU	MTPU	LPU
Halstead per line (ITL)	1.00	.178	.678	.194	.009
McCabe per line (MTL)		1.00	-.046	.056	.919
Halstead per unit (ITPU)			1.00	.598	-.060
McCabe per unit (MTPU)				1.00	-.199
Lines per unit (LPU)					1.00

TABLE 5: DENSITY CORRELATIONS

5. FURTHER RESEARCH

The metrics analyzer HUMAN has been applied to most of the files in the Ada Repository. Data collection will be obtained for the remaining large files as soon as an Ada compiler with larger capacity is obtained; the intended machine is a SUN 3 with 8 MB of main memory and a disk of 327 MB. The results should be available by the time that this paper appears, assuming that the Ada source code of HUMAN has been written in a completely portable manner. The complete analysis will include the Halstead and McCabe analysis of all of the Ada source code files in the repository. This will provide a benchmark for analysis of programmer style, as well as being a starting point for other "software quality" metrics.

The future metrics analysis work will include measurements of other factors such as lists of possible execution paths, deadlock determination for programs with tasking, development of useful metrics for estimating reusability and portability of packages. We expect to incorporate this work into a project which will provide an analysis of the code at the time of parsing into separate tokens since our experience with other metrics analysis programs indicates that storing such information at one time and making several passes through the list of tokens will make the determination of these additional metrics for portability, reusability, path analysis, potential deadlock, etc. much easier. Results obtained will be easily replicatable since the Ada Repository is easily available and does not consist only of proprietary information.

REFERENCES

1. Halstead, M.H., *Elements of Software Science*, Elsevier North-Holland, New York, 1977.
2. Kearney, J.K., Sedlmeyer, R.L., Thompson, W.B., Adler, M.A., and Gray, M.A., *Problems with Software Complexity Measurement*, Proc. 1985 ACM Computer Science Conference, pp340-347, Cincinnati, March 12-14, 1985.
3. Keller, S.E., and J.A. Perkins, *An Ada Measurement and Analysis Tool*, Proc. Third Annual National Conference on Ada Technology, p188-196, Houston, March 20-21, 1985.
4. Leach, R.J., *Ada Software Metrics and Their Limitations*, Proc. Joint Ada Conference, p285-293, Washington, D.C., March 16-19, 1987.
5. McCabe, T.J., *A Complexity Measure*, IEEE Trans. Softw. Engr., SE-2, p308-320, Dec. 1976.
6. Perkins, J.A., Lease, D.M., and Keller, S.E., *Experience Collecting and Analyzing Automatable Software Quality Metrics for Ada*, Proc. Fourth Annual National Conference on Ada Technology, p67-74, Atlanta, March 19-20, 1986.
7. Reynolds, R.G. and D. Roberts, *PARTIAL: A Tool to Support the Metrics Driven Design of Ada Programs*, Proc. 1985 ACM Computer Science Conference, p213-219, Cincinnati, March 12-14, 1985.
8. Taylor, R.N. and T.A. Standish, *Steps to an Advanced Ada Environment*, IEEE Trans. Softw. Engr., SE-11, p302-310, March, 1985.
9. Woodfield, S.N., Shen, Y.Y., and H.E. Dunsmore, *A Study of Several Metrics for Programming Effort*, J. Systems and Software, vol2, 97-103, 1981.

BIOGRAPHY

Ronald J. Leach is a Professor in the Department of Systems and Computer Science at Howard University. He has B.S., M.A., and Ph.D degrees in Mathematics from the University of Maryland at College Park and an M.S. degree in Computer Science from Johns Hopkins. His research interests include computer graphics, analysis of algorithms, user interfaces, and software engineering (especially software metrics and Ada programming).

Ada Implementation of Sequential Correspondent Operations for Software Fault Tolerance

Pen-Nan Lee

Aderbad Tamboli

Department of Computer Science
University of Houston
Houston, Texas 77004.

Abstract

Based on the concept of Correspondent Computing, we propose "Sequential Correspondent Operations" as a strategy for the sequential implementation of software fault tolerance, providing redundancy through the use of correspondent operations. Error detection and forward error recovery are performed by the use of a Comparative test. The inherent high-level programming facilities provided by Ada, make it the language of choice for implementing Sequential Correspondent Operations. This sequential scheme is the most natural approach for systems with limited hardware resources. The use of Ada enhances the power of Sequential Correspondent Operations, making it an effective and feasible strategy for developing fault tolerant software.

1. INTRODUCTION

As computers have grown more complex, not only in their internal structure, but also in their applications, the need to ensure reliability has increased. In fact computers are increasingly being used in critical application areas where loss of computing power for even a few seconds can be disastrous. To ensure the proper functioning of the computer system even in the face of faults has become a challenge.

There are two complementary approaches to constructing highly reliable systems. The first is fault prevention, which tries to ensure that the implemented system will not contain faults. This can be achieved by using structured design methodologies, quality control etc. to avoid introducing faults in the system. Testing and other validation techniques are also used to find and remove errors. But as the complexity of the system increases, residual faults exist despite extensive application of fault prevention techniques. To overcome these residual design inadequacies or faults, fault tolerant techniques are applied to ensure the reliability of the system.

Fault tolerance may be defined as the ability to detect and recover from residual design inadequacies (errors) without any appreciable loss in either computation or time. There are a number of fault tolerant strategies available for implementing reliable systems, prominent among which are the Recovery Block Scheme [7] and the N-Version Programming [2]. Recently, the authors have proposed a fault tolerant strategy based on the power of Correspondent Computing [6].

This paper discusses the sequential aspects of Correspondent Computing. In section 2, we present a brief background of the concepts of Correspondent Computing, including its error detection mechanism the Comparative test. Section 3 presents a sequential strategy "Sequential Correspondent Operations" based on Correspondent Computing

for implementing fault tolerant software. An Ada implementation of Sequential Correspondent Operations is shown in section 4. Finally, in section 5 we conclude the discussion and give directions for future research.

2. CORRESPONDENT COMPUTING

Correspondent Computing [5,6] is based on the philosophy of correspondence which can be stated as: "if executing an operation can produce a significant effect, then another equally significant effect can be generated by another semantically correspondent operation". Correspondence is the property that binds two operations together. The relationship (between the two operations) being reciprocal, equivalent or analogous. If a distinct and precise relationship exists between two operations, and the results (or effects) of these two operations also exhibit this same precise and distinct relationship, then these two operations have a correspondence relationship. Operations having a correspondence relationship are termed as Correspondent operations.

Let us illustrate the concept of Correspondent Operations with an example. Consider an object "A" resting on the ground. It is now exerting a force on the ground equal to its weight, let this be W_1 . To maintain equilibrium, the ground is exerting an equal and opposite reaction R_1 . Now let us place another object "B" on the ground exerting a force equal to its weight W_2 , and the reaction from the ground being R_2 . Since both objects are in equilibrium, the weights are equal to the reactions, i.e. $W_1 = R_1$ and $W_2 = R_2$. In addition, observe that there exists a precise and distinct relationship between weights of the two objects (W_1 and W_2) and this same relationship also defines the reactions (R_1 and R_2), i.e. $(W_1/W_2) = (R_1/R_2) = \text{constant}$. Thus the two objects are correspondent with one another. Using the same analogy, two operations are correspondent operations if the operations and the results (of the two operations) demonstrate the same relationship.

A program is structured into units (modules, functions etc.), and each of which can be regarded as an operation. An operation consists of sequences of smaller operations, the smallest being the single instruction. The operation whose effect is the target of correspondent computing is called the primary operation. The primary operation can be thought of as an operation whose correct execution is more critical than other constituent program operations, and hence is the basis for the creation of redundant operations. In a non-redundant environment a program consists of only primary operations. Operations which generate effects that correspond to the primary operation are called correspondent operations. The effects of a correspondent operation may be equivalent, complementary, contradictory, or competing to the effects of the primary operation. Correspondent operations can be categorized as Reciprocal, Duplicate, or Residual. Reciprocal

operations are semantically the inverse of the primary operation. Duplicate operations are those whose behavior is semantically equivalent to the primary operation. Residual operations are those that exhibit correspondence relationships, but are neither exactly duplicate, nor reciprocal operations.

All fault tolerance must be based on the provision of useful redundancy, both for error detection and error recovery. In software the redundancy required is not a simple replication of programs but redundancy of design [7]. Correspondent computing provides the requisite redundancy through the use of correspondent operations for each primary operation. In addition to providing useful redundancy, these correspondent operations are also powerful enough to assist in error detection and error recovery.

Error detection is accomplished by the use of a Comparative Test. The specifications of the Comparative test are based on the precise, predefined and distinct relationship of the primary and its correspondent operations. As has been already stated, for operations to be considered "correspondent operations" they must exhibit the same relationship for the effects, as for the operations. That is, if two correspondent operations produce two effects (results) then these results must have the same relationship as the operations. The only possible circumstance in which the relationship of the two results differ, is an error occurring during execution of one of the operations, thereby changing the result of one of the operations from the desired result. The Comparative test is based on this principle, and performs error detection by observing if the relationship of the results differ from the predefined relationship of the two correspondent operations. For example, to check the results of a sorting operation, a Comparative test can be used. The comparative test works by comparing the results of the primary sort operation, and the residual sort operation, a simple match procedure described below can detect if errors have occurred.

```
for I in 1..ARRAY_SIZE loop
  if P(I) /= A (RE(I)) then
    MATCH := FALSE;
  end if;
end loop;
```

This simple procedure is powerful enough to detect any data inconsistency errors as well as computation errors that might have occurred in either the primary or the residual operation.

The error detection component is incomplete without some form of error recovery to achieve fault tolerance. Correspondent computing provides a forward error recovery strategy, avoiding the high time-space overhead in maintaining mechanisms for rolling back the system state to an error free state. Again, the redundant correspondent operations provide the requisite component for error recovery. When the comparative test detects an error, the erroneous result is masked and another correspondent operation is initiated, and on termination its results tested with the results of the previously executed operations. This ensures, that the correct result is always obtained.

In the next section, we shall discuss in detail the sequential aspects of Correspondent Computing.

3. SEQUENTIAL CORRESPONDENT OPERATIONS

The use of Correspondent Computing for fault tolerant systems is based on the redundancy provided by the correspondent operations. These correspondent operations are not only simple to formulate, but, are also powerful enough to

assist in error detection and error recovery. We present "Sequential Correspondent Operations" as a strategy for implementing fault tolerant software. This sequential scheme is the most natural approach for systems with limited hardware resources.

A program consists of a sequence of operations, and these operations consisting of sequences of smaller operations. In a fault tolerant environment, it is necessary to provide for error checking, but obviously error checking for each basic operation is too expensive, in terms of both time and space complexity to perform. From the viewpoint of software fault tolerance, the correct behavior of some operations is more critical than others. Therefore, it is sufficient to provide redundancy for these critical operations, rather than for every basic operation. Keeping this objective in mind, we assume that a program consists of two types of operations, critical operations requiring redundant components, and non-critical operations. These critical operations having error detection and recovery capabilities will be formed using Sequential Correspondent Operations.

A fault tolerant program, similar to any other program, is composed of a number of sub-program units, such as modules, procedures, functions etc. Using Sequential correspondent operations, a program is composed of conventional (non-redundant) units, and sequential correspondent operations. These units are again composed of a number of conventional and redundant sub-units. Simply put, a program contains a number of nested non-redundant (conventional), and redundant (sequential correspondent operations) units.

3.1 Structure of Sequential Correspondent Operation

Sequential Correspondent operations (SCO) consist of three components, 1) entry condition : it is the initiation of the sequential correspondent operation, 2) the processing component : consisting of the primary, its correspondent operations and the comparative test, and 3) the exit condition : the termination condition where the results are returned to the calling module. The structure of a SCO is shown in figure 1.

A sequential correspondent operation is initiated when the preceding module completes, and passes control to the entry condition of the sequential correspondent operation. The execution sequence is :

- 1) Execution of the primary operation.
- 2) Execution of the first correspondent operation.
- 3) The results of the primary and correspondent operation are sent to the comparative test, to decide if the results match.
- 4) If the results match, the sequential correspondent operation terminates.
- 5) If the results fail to match, the next correspondent operation is executed. This new result, along with the results of the primary and the previous correspondent operations is sent to the comparative test, to determine if a match has occurred. Step 5 is repeated until a match occurs, or all correspondent operation are terminated.

The use of entry and exit conditions enables easy expansion of the system. For example, at a later stage, a conventional component can be replaced with a non-redundant component without any interfacing problems.

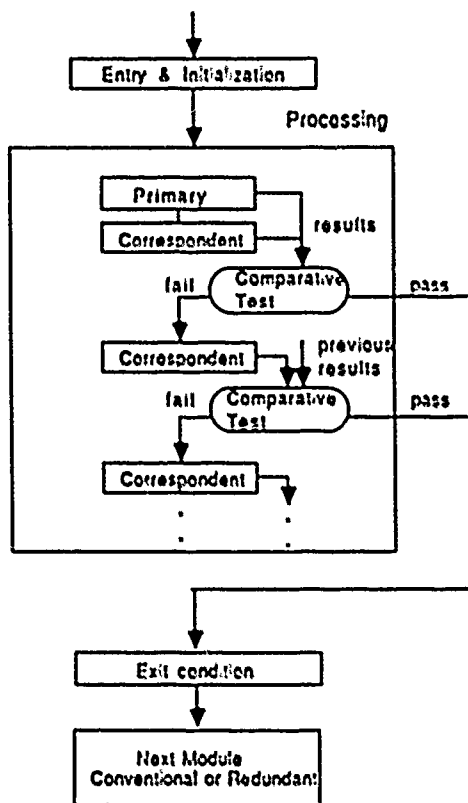


Fig. 1 : Structure of a Sequential Correspondent Operation

3.2 Error Detection and Error Recovery

A sequential correspondent operation consists of a primary, one or more correspondent operations, and a comparative test. The execution of the sequential correspondent operations begins with the primary operation being initiated. After the primary operation completes execution, one of the correspondent operations is executed next. On completion of the first correspondent operation (C1) the comparative test is initiated. The comparative test determines if the results of the primary and the first correspondent operations match. If a match is obtained, that is, no error has been detected, the result of the primary (or the correspondent) operation is returned to the calling module.

When a match fails, that is an error state is detected in the result, another correspondent operation (C2) is initiated. On completion of this correspondent operation, the comparative test is again invoked. The comparative test now compares the result of the second correspondent operation (C2), with the results of the primary and the first correspondent operation (C1). If a match occurs, the comparative test selects the correct result (from the two results that match) and returns control to the calling routine. If no match occurs, another correspondent operation (C3) is initiated, and on completion its result is used by the comparative test to determine if the correct result can be obtained. This process is repeated until either the correct result is obtained, or all the correspondent operations are completed.

When none of the primary, or the correspondent operations in a sequential correspondent operation match, that is, the complete unit fails, another sequential correspondent unit (operation) might be invoked. In a nested sequential correspondent operation, consisting of a primary and one or more correspondent operations, each of which (primary and correspondent operation) is a sequential correspondent operation. The failure of a lower level sequential correspondent operation, (which might be the primary operation of the higher level unit) the higher level sequential correspondent operation would invoke another lower level sequential correspondent operation (one of the correspondent operations of the higher level Sequential correspondent operation), repeating the process until the correct result is obtained. The number of redundant units, and the number of correspondent operations per unit, for each primary operation depend on the criticality of the primary operation. Obviously, a highly critical operation would require a significantly higher degree of redundancy, than a non-critical operation.

The comparative test not only detects errors, but also assists in error recovery. This error recovery consists of ensuring the correct result is passed to the calling module. When the comparative test receives the result from the primary and the correspondent operations, it makes a decision on whether the result is correct. At this point, a correct result is determined if any two results match. Whenever, the correct result is not obtained from the primary operation, it needs to be converted to the primary result before being returned to the calling routine. This problem can be solved by using "reverse correspondent operations". Reverse correspondent operations are operations that convert the result of a correspondent operation to an "equivalent primary result". For example, if the primary operation is to sum an array of integers, and its result is equal to 100. Its reciprocal correspondent operation is to subtract the array elements (instead of adding), giving a result equal to -100, then equivalent primary result for the reciprocal operation can be obtained by negating the reciprocal result.

The use of reverse correspondent operations, and equivalent correspondent operations make the design, implementation and execution of the comparative test easier. In addition, the result from the correspondent operations does not need to be converted by the comparative test. However, there is a slight drawback with this scheme, the reverse correspondent operations require additional execution time, which might slow down the entire sequential correspondent operation.

The use of sequential correspondent operations enhances the ease of implementation of a fault tolerant program. In the next section we will give an Ada implementation of sequential correspondent operations for fault tolerant applications.

4. ADA IMPLEMENTATION OF SEQUENTIAL CORRESPONDENT OPERATIONS

Sequential Correspondent Operations is an effective strategy for implementing fault tolerant software, providing redundancy through the use of correspondent operations. Ada, the new general purpose programming language, is based on the definitions proposed by the U.S. Department of Defense for use in embedded systems. It is the culmination of a decade of specification and revision of successive versions of the language, and reflects the current trends towards data abstraction, multi-tasking, generics, exception handling, readability, reliability, etc. Ada is a powerful, yet expressive and feature rich language, which addresses a wide range of

application areas. It is a tool that encourages good software engineering principles, having features to detect more errors early and automatically, helping programmers to write good programs, without inhibiting creativity and ingenuity. The use of Ada enhances the power of Sequential correspondent operations, making its development and implementation easier, faster and more efficient.

A fault tolerant program consists of number of nested levels, each level containing both redundant and non-redundant modules. In Sequential Correspondent operations, a program comprises of a number of nested conventional and redundant (sequential correspondent operations) modules. An Ada implementation of such a program could be written as :

```

procedure MAIN is
  procedure CONVENTIONAL_MODULE_1 is
    .
  procedure SEQUENTIAL_CORRESPONDENT_OPERATION_1 is
    .
  procedure SEQUENTIAL_CORRESPONDENT_OPERATION_1 is
    .
  procedure CONVENTIONAL_MODULE_1 is
    .
  procedure CONVENTIONAL_MODULE_N is
    .
  procedure SEQUENTIAL_CORRESPONDENT_OPERATION_N is
    .
begin -- procedure MAIN
  .
and MAIN ;

```

The above example shows the skeleton of a fault tolerant program using sequential correspondent operations. Of course, each of the conventional modules and the Sequential correspondent operations (modules) can be further nested, each comprising of both conventional modules and sequential correspondent operations. Considering only the sequential correspondent operation at the lowest level, that is, there are no more nested redundant modules inside this sequential correspondent operation. The structure of such a sequential correspondent operation could be implemented as

```

procedure SEQUENTIAL_CORRESPONDENT_OPERATION_1 is
  -- local variables
  procedure SEQ_CORR_OPERATION_1_PRIMARY is
    begin
      .
    end SEQ_CORR_OPERATION_1_PRIMARY ;
  procedure SEQ_CORR_OPERATION_1_RECIPROCAL is
    begin
      .
    end SEQ_CORR_OPERATION_1_RECIPROCAL ;

```

```

  procedure SEQ_CORR_OPERATION_1_DUPLICATE is
    begin
      .
    end SEQ_CORR_OPERATION_1_DUPLICATE ;
    -- other correspondent operations (as needed)
    .
begin -- SEQUENTIAL_CORRESPONDENT_OPERATION_1
  SEQ_CORR_OPERATION_1_PRIMARY (DATA) ;
  SEQ_CORR_OPERATION_1_RECIPROCAL (DATA) ;
  COMPARATIVE_TEST ;
  if not (MATCH) then
    SEQ_CORR_OPERATION_1_DUPLICATE (DATA) ;
    .
  end if ;
end SEQUENTIAL_CORRESPONDENT_OPERATION_1 ;

```

The sequential correspondent operation shown above consists of a primary, n ($n > 1$) correspondent operations and a comparative test. Execution starts when control is passed to the sequential correspondent operation SEQUENTIAL_CORRESPONDENT_OPERATION_1. First, the primary operation (in this case SEQ_CORR_OPERATION_1_PRIMARY) is executed. Next one of the correspondent operations is initiated. On completion of the first correspondent operation (SEQ_CORR_OPERATION_1_RECIPROCAL), the Comparative test is started. The comparative test, obtains the results from the two completed operations, and checks if the two results match. If the match occurs, then the results are correct. When a correct result is obtained, the results are passed to the calling routine, and the sequential correspondent operation terminates. However, if a match has not occurred, then another correspondent operation is initiated, the process being repeated until the correct results are obtained, or all correspondent operation are completed.

Using this basic structure, let us elaborate the power of sequential correspondent operation with an example. The example consists of a fault tolerant sorting of an array of integers. This simple program consists of reading the array of integers, sorting this array in an ascending order, and printing the result of the sorted array. In this problem, the sort routine is critical and therefore requires redundancy. Hence, the sort will be fault tolerant, and implemented using Sequential Correspondent Operations. To simplify the coding and to improve readability the implementation details will be ignored.

```

procedure SORT_MAIN is
  procedure READ_ARRAY (ORIG_ARRAY : out ARRAY_TYPE) is
    begin
      .
    end READ_ARRAY ;
  procedure SEQ_CORR_SORT (ORIG_ARRAY : in ARRAY_TYPE ;
    RESULT : out ARRAY_TYPE) is
    begin
      .
    end SEQ_CORR_SORT ;

```

```
procedure PRINT_ARRAY (RESULT : in ARRAY_TYPE) is
begin
```

```
end PRINT_ARRAY;
```

```
begin -- SORT_MAIN
  READ_ARRAY (OR_ARRAY);
  SEQ_CORR_SORT (OR_ARRAY, RESULT_ARRAY);
  PRINT_ARRAY (RESULT_ARRAY);
end SORT_MAIN;
```

Since the reading and printing are conventional modules, we will not be showing their implementation. The sequential correspondent operation SEQ_CORR_SORT is described next.

```
procedure SEQ_CORR_SORT (ORIG_ARRAY : in ARRAY_TYPE;
  RESULT : out ARRAY_TYPE) is
```

```
-- local variable declarations
```

```
-- primary and correspondent sorting procedures
procedure PRIMARY_SORT (O_ARRAY : in ARRAY_TYPE;
  P_ARRAY : out ARRAY_TYPE) is
```

```
begin
```

```
-- primary sorting
```

```
end PRIMARY_SORT;
```

```
procedure RECIPROCAL_SORT (O_ARRAY : in ARRAY_TYPE;
  R_ARRAY : out ARRAY_TYPE) is
```

```
begin
```

```
-- reciprocal sorting
```

```
-- convert results to equivalent primary result
```

```
-- using reverse correspondent operations
```

```
end RECIPROCAL_SORT;
```

```
procedure DUPLICATE_SORT (O_ARRAY : in ARRAY_TYPE;
  D_ARRAY : out ARRAY_TYPE) is
```

```
begin
```

```
-- duplicate sorting
```

```
end DUPLICATE_SORT;
```

```
procedure RESIDUAL_SORT (O_ARRAY : in ARRAY_TYPE;
  RE_ARRAY : out ARRAY_TYPE) is
```

```
begin
```

```
-- residual sort
```

```
-- convert results to equivalent primary result
```

```
-- using reverse correspondent operations
```

```
end RESIDUAL_SORT;
```

```
-- other correspondent operations if desired
```

```
procedure COMPARATIVE_TEST (RES_IN : in ARRAY_TYPE;
  RES_LIST : in LIST_TYPE;
  SIZE : in INTEGER;
  MATCH : in out BOOLEAN) is
```

```
begin
```

```
MATCH := FALSE;
```

```
for I in 1..SIZE loop
```

```
  MATCH := TRUE;
```

```
  TEMP := RES_LIST (I);
```

```
    for J in 1..ARRAY_SIZE loop
      if RES_IN (J) /= TEMP (J) then
        MATCH := FALSE;
```

```
      end if;
```

```
    end loop -- for J
```

```
    if MATCH then
```

```
      RESULT := RES_IN;
```

```
    end if;
```

```
  end loop
```

```
end COMPARATIVE_TEST;
```

```
begin -- SEQ_CORR_SORT
```

```
  PRIMARY_SORT (ORIG_ARRAY, P_ARR);
```

```
  RESULT_LIST (LIST_SIZE) := P_ARR;
```

```
  RECIPROCAL_SORT (ORIG_ARRAY, R_ARR);
```

```
  COMPARATIVE_TEST (R_ARR, RESULT_LIST,
    LIST_SIZE, MATCH);
```

```
  if not (MATCH) then
```

```
    LIST_SIZE := LIST_SIZE + 1;
```

```
    RESULT_LIST (LIST_SIZE) := R_ARR;
```

```
    DUPLICATE_SORT (ORIG_ARRAY, D_ARR);
```

```
    COMPARATIVE_TEST (D_ARR, RESULT_LIST,
      LIST_SIZE, MATCH);
```

```
  if not (MATCH) then
```

```
-- other correspondent operations
```

```
  end if;
```

```
end SEQ_CORR_SORT;
```

Any sorting algorithm can be used as the primary sort algorithm. Using this primary sort algorithm, its correspondent operations are implemented using the correspondent relationships, such as duplicate, reciprocal and residual. In this example, we are converting the results of the correspondent operations to an equivalent primary result, so that the comparative test becomes easier. The results from the primary and the correspondent operations are stored in a list (RESULT_LIST). The comparative test then compares the results in the RESULT_LIST with the new result obtained. In this way each result can be compared with every other result. When a match is obtained, the result is taken as the correct result and returned to the calling routine, which in this program is procedure SORT_MAIN.

5. CONCLUSION

This paper presents a fault tolerant strategy based on Correspondent Computing. The strategy Sequential Correspondent Operations consists of a primary, "n" (>= 1) redundant correspondent operations, and a comparative test. These correspondent operations not only provide the desired redundancy for fault tolerant applications, but are also powerful enough to assist in error detection and error recovery.

In this scheme, a set of primary and correspondent operations (SCO) are executed sequentially. In each SCO, first the primary and one correspondent operation are executed and their results passed to the comparative test. The comparative test performs error detection by comparing the results of the two operations. If the results match, that is, no error has occurred, the result is passed as the correct result. However, if an error has occurred, then another correspondent operation is initiated, and its result passed to the comparative test. The comparative test now compares this result with the two previous results, to determine if an error has occurred. This process continues until either the correct result is

obtained, or all correspondent operations are completed.

This sequential scheme is the most natural approach for systems with limited hardware resources. The use of Ada, with its inherent high-level programming facilities, make it an elegant, effective and feasible strategy for development and implementation of fault tolerant software.

REFERENCES

- [1] Anderson, T. and Lee P., "Fault tolerance terminology proposals", 12th Annual International Conference on Fault Tolerant Computing, June 1982.
- [2] Chen, L., and Avizienis, A., "A fault-tolerance approach to reliability of software operations", 8th Annual International Conference on Fault Tolerant Computing, June 1978.
- [3] Gehani, N., "Ada: An Advanced Introduction", Prentice-Hall, New Jersey, 1984.
- [4] Lee, Pen-Nan., "Implementing efficient software fault tolerance using concurrent Ada," Proceedings of ACM South Central Regional Conference, Lafayette, Louisiana, Nov. 1987.
- [5] Lee, Pen-Nan., "Correspondent Computing," Proceedings of ACM 1988 Computer Science Conference, Atlanta, Georgia, Feb. 1988.
- [6] Lee, Pen-Nan, Tamboli, Aderbad and Blankenship, Jeff., "Correspondent Computing based Software Fault Tolerance", Proceedings of the 26th Allerton Conference on Communications, Control and Computers, Urbana, Illinois, Sept 1988.
- [7] Randell, B., "System structure for software fault tolerance", IEEE Transactions on Software Engg. Vol SE 1, No. 2, June 1975.
- [8] U.S. Department of Defense, "Programming Language Ada: Reference Manual," Vol 106, Lecture Notes in Computer Science. Springer-Verlag, New York, 1981.
- [9] Wallace, Robert., "Practitioners Guide to Ada," MacGraw-Hill, New York, 1986.

The Ada Binding for POSIX

Stowe Boyd

David Emery

Terry Fong

Mitch Gart

Abstract

The Technical Committee on Operating Systems of the IEEE Computer Society sponsors the P1003 Committee's efforts on the Portable Operating System Interface for Computer Environments (POSIX). The P1003.5 Working Group was formed in 1987 to develop a POSIX Ada interface. This report provides the rationale for the binding, details specific problems encountered and their solution, and outlines the status of the effort.

Introduction to POSIX

There is a growing convergence toward the Unix operating system as a standard for time-sharing and single-user computing environments. The wide variety of Unix-derived systems — especially System V and Berkeley derivatives — has made the promise of a standard Unix quite elusive.

The IEEE has formed a committee to develop a Portable Operating System Interface Standard (POSIX). The work of this group, IEEE P1003, has grown from the early efforts of *lusr/group*, and the influential Standard that their technical committee proposed. In 1985 the two groups merged, and immediately formed a working group — P1003.1 — to define a programming language interface for the proposed standard.

"The goal of the P1003.1 Working Group was to promote portability of application programs across Unix environments by developing a clear, consistent and unambiguous standard for the interface specification of a portable operating system based on the Unix system documentation."¹

At the time of this writing the POSIX interface specification is in ballot. It was not until mid-1987 that the P1003.5 committee was formed², charged with specifying an Ada binding for the then-proposed draft 12 version of the POSIX interface specification. Several other working groups have been formed to deal with other pressing requirements, such as real-time, and security.

In the next section, a brief overview of the barriers to a POSIX Ada is presented, followed by a short rationale of the group's approach. A short summary and status report can be found at the end.

Barriers to Ada Binding

The Base Specification

The basic POSIX document consists of a collection of C function declarations, with supporting header files and text to explain the semantics of each call. Therefore, the POSIX interface is specified with C semantics, which are not the same as Ada semantics. One problem with C is that it is not a strongly typed language. Many different POSIX routines have "int" parameters. One of the challenges of doing an Ada binding is trying to determine, for a given POSIX/C int parameter, an associated Ada type. This is particularly true for flag values, which map most directly to Ada enumeration types.

There is some support for modularity in the C interface via the "header" files. These tend to encapsulate common type declarations and constants used by a set of related C functions. However, a 1-to-1 mapping of header files to Ada packages will not work, because many of the POSIX/C functions require more than 1 header file, but an Ada operation can be defined in only 1 package.

The C Bias

There are many places in the POSIX definition where the POSIX/C binding refers back to the definition of the C lan-

language. Perhaps the most notable example concerns memory management. There is no POSIX routine to allocate memory. Instead, memory is allocated using the C malloc() routine, which is defined by the appropriate C standard (ANSI C, in this case), and by the particular implementation of C used for a specific POSIX implementation.

Another example of where the POSIX definition defers to the C language is in system limits. POSIX depends on the C implementation to define such things as INT_MAX, INT_MIN and CLK_TCK. These correspond to System.Max_Int, System.Min_Int and System.tick. Since there is no POSIX-defined minimum values for these numbers (such definitions are in the C Language Standard), the POSIX Ada binding can make no particular assumptions about system limits. This is particularly important for CLK_TCK, which is really a function of the hardware and operating system's clock resolution, rather than the capabilities of a given compiler.

Style Conflict

There are several inherent conflicts between C 'style' and Ada 'style'. C defines no language error signalling facility, so each C function has its own way of indicating an error. In many cases, the function returns a known value for an error (such as an address of 0), and another value for success. This is very hard to directly map to Ada. Instead, Ada exceptions should be used, but then many common Unix coding practices that call a routine and branch to an error handler based on the result, would have no direct Ada analog.

One particular C coding technique used often in the POSIX C specification is a function that both returns a value as a function result and also changes one of the function's parameters. The direct Ada analog is a function with IN OUT or OUT parameters, which is not allowed in Ada. In many cases, the single C operation really performs several logically distinct functions, and the mapping from this functionality to Ada is not always obvious.

Minimal Interface Definition

Another problem with the POSIX definition is that POSIX is a minimal interface definition. In order to allow implementation freedom and to maintain compatibility with several existing systems, many POSIX calls define a

minimum set of services or data elements. For instance, the data type that defines a directory entry contains 'at least the following fields: ...'. Again, this tends to conflict with the Ada strong typing model, where each element of a record type must be known at compile time. Furthermore, there is the issue of name conflicts that can occur when an implementation-defined field on some system has the same name (visible in the same scope) as some programmer-defined name. The program may compile without error on one POSIX implementation, but will not compile on another implementation.

Part of this minimal interface consists of a set of predefined names and constants that define a specific POSIX environment. For instance, there is a POSIX constant named PATH_MAX, which is the maximum number of characters (bytes) in a pathname. The definition in Ada of a type for pathnames (which should have up to PATH_MAX characters) is not obvious. String handling in general is a non-trivial problem, because of the much more dynamic C definition of strings, and also because POSIX permits 8 bit characters, which are not part of the Ada predefined CHARACTER type. So, what type should the POSIX operation that provides the name of the current working directory (the C getcwd() function)? How do you note that this value can be only PATH_MAX bytes long, and how do you handle the case where some value in the result has a byte value > 127, which is not a legal value for an element in the Ada predefined STRING type?

Tasks versus Processes

Finally, POSIX system calls assume a single thread of control. Since an Ada program can contain many tasks (possibly running concurrently on a multiprocessor implementation), the specific effects of concurrency and preemption must be studied for each POSIX interface. Some services present no problems, but others, particularly those that maintain local state such as I/O services, require some sort of redefinition to define their behavior in the face of multiple threads of execution.

POSIX Signals are the worst case for an Ada binding supporting tasking. In POSIX, a signal is an asynchronous event, and the user can associate a function to handle the arrival of a signal. Some signals represent Ada exceptional conditions, such as numeric errors or attempts to dereference a null pointer. Other signals represent true

asynchronous events, such as the completion of I/O. In a multi-threaded Ada program, should a signal be delivered to a specific task, or to all tasks? How can a task or task entry or subprogram be associated with a given POSIX signal? An Ada binding for POSIX must support all of the POSIX services, including POSIX signals. Developing the correct model for signals will be the hardest part of the Ada binding.

Rationale for the Ada Binding

Goal

The principal goal of the working group is to create a POSIX Ada binding which is useful, portable, and which exhibits good style. POSIX primitives must be made available to Ada programs such that POSIX Ada applications can be specific to POSIX, but otherwise portable across machine architectures and Ada compilers.

Level of Abstraction

Three possible levels of abstraction for POSIX Ada have been identified: direct, abstract, and independent. A direct binding is one where the base, C operations are mapped as closely as possible into Ada. An abstract binding defines POSIX abstract data types by abstraction from the base definition, and then partitions these abstract types and their operations into logically related collections³. An independent binding goes farther in the abstraction process; there is no strict one-to-one relationship between POSIX C and Ada operations.

In committee, the abstract level has been accepted both in principle and in practice. It is clear that POSIX Ada will comprise a set of Ada components which represent the functionality of POSIX, organized to provide good Ada style while remaining intelligible to a knowledgeable POSIX C user.

Any POSIX C function will be subject to the following conversion rules when being mapped to POSIX Ada:

- Functions will be bundled in Ada packages as subprograms, task entries, or generic units.
- Use of Ada identifiers will lead to significantly more expressive names.
- Polymorphic functions will often be decomposed into a family of overloaded subprograms.

- Some functions may be omitted where inappropriate for Ada (e.g. malloc()).
- Error states will lead to termination by exception propagation, rather than the setting of a return code.

Easy and Hard Mappings

Some aspects of the binding are relatively easy, since there appears little conflict between POSIX C functionality and a straight-forward abstract Ada equivalent. In these cases the main effort is choosing identifiers, parameter types, and error handling mechanisms. Examples include POSIX I/O, directory, and system database primitives.

Other aspects of the binding are not as amenable to this simple transformational approach; in particular, difficulties arise from the relationship between POSIX processes, Ada tasks and programs. POSIX signals are semantically difficult for a number of reasons, treated in the previous section.

It should be noted that it is precisely those POSIX operations which are the easiest to map into Ada which are likely to be the most useful. A majority of applications will most likely rely only upon the "easier" file and directory operations, while only a small minority will need to directly manipulate signals.

The Basics: Names, Types, and Error Handling

A surprisingly general agreement has been reached in the areas of names, types, and error handling. Expressive Ada identifiers will be used in place of cryptic (if not misleading) C identifiers⁴. Renaming to provide C equivalent names has been proposed by some, but this is controversial.

The basic POSIX STRING and FILE_DESCRIPTOR types — which are used throughout the binding — have received a great deal of the group's attention. Since the Ada character set is limited to 7-bit ASCII, while POSIX requires both 8-bit and multi-byte characters, most operations will rely upon a POSIX STRING type. This type will be interconvertible with respect to Ada Standard STRING⁵. FILE_DESCRIPTOR will be either an explicit integer type, or a private type; arguments for integer type are based upon the need to index arrays by FILE_DESCRIPTOR.

One significant departure from base POSIX is the setting of "flags" parameters — bit strings — by application of Ada constructors in place of bit value assignment.

Exceptions are the basic mechanism for POSIX Ada "error handling." Operations which encounter an exception condition will raise an expressive and specific exception, rather than returning a "strange value." The degree to which other mechanisms will be supported is still unresolved; there is some agreement on a "predicate" style for certain sorts of POSIX access.

Conclusions and Status

The authors believe that the development of a single, widely used operating system interface in Ada will have immense benefits for all users of the language. Ada source code reuse has not proven to be as significant as hoped; one major barrier may be the lack of a common substrate for reusable Ada. POSIX Ada can be such a foundation.

The POSIX Ada working group has held three meetings, each of two or three days. A fourth meeting is planned for June in Houston, sponsored by MITRE Corporation, with meetings planned with NASA representatives and contractors ⁶. A sixth meeting is scheduled for October 1988, and a preliminary draft of a POSIX Ada interface is planned for early 1989.

-
- 5 Any issues relative to ASCII.NUL will be handled by the interface.
 - 6 NASA is considering POSIX for the Space Station project host and target architecture.

¹ POSIX Explored, /usr/group, Santa Clara California.

² Note that the authors are officers of the P1003.5 working group:

T. Fong (US Army) — Chair

S. Boyd (Meridian) — Co-Chair

D. Emery (MITRE) — Secretary

M. Galt (Alsys) — Rationale Editor

³ This approach is quite similar to that described by H. Fisher, in his "PCTE Ada Interface" presentation, APSE Builder's Working Group, SIGAda International Conference, November 1987.

⁴ Although names should not become unwieldy.

Ada SUMMER SEMINAR--TEACHING THE TEACHERS

Dr. M. Susan Richman
Penn State Harrisburg

Dr. Charles G. Petersen
Mississippi State Univ.

Mr. Donald C. Fuhr
Tuskegee University

ABSTRACT

College and university faculty have different needs and expectations from a programming language course than most other audiences, especially when Ada is the language. Specifically, they require a grounding in software engineering, adequate time for assimilation of the concepts involved, and hands-on experience working on a large project. These requirements are not met by the typical college course, short course, or self-study.

The Ada Curriculum Development Seminar described herein more than met its objective of providing participants the background to enable effective integration of Ada into the curricula of their home institutions.

This paper discusses the design of the program, the challenges presented by the participants' individual backgrounds, and the special techniques used in the presentation of the information. Details of the laboratory exercises and team solutions are discussed and evaluated. Participant reactions are assessed and implications for future program design derived. General information for other institutions contemplating similar programs is included.

INTRODUCTION

Program Background

The Ada Curriculum Development Seminar held at Tuskegee University from June 5 through July 1, 1988 had its roots in four years of previous similar programs. These programs were sponsored by the U.S. Army Center for Tactical Computer Systems (CENTACS) and were held at Ft. Monmouth, New Jersey during the summers of 1981, 1982, and 1983, and at Tuskegee in 1984. The objectives of all these programs were to propagate the Ada programming language into college and university computer science curricula by providing an intensive learning experience for faculty members. All three of the professional staff of this Seminar were participants in one of the previous programs.

Seminar Objective

The seminar objective as stated in the application brochure was "to provide comprehensive experience in the Ada Programming Language to college and university computer science faculty so that they may, in turn, effectively teach Ada to their students. In the course of the seminar, the

expertise and experience of the participants will be applied to curriculum design and the development of materials for use in courses at their home institutions. All of the features of the Ada Programming Language will be explored, and extensive hands-on experience will be provided."

The seminar goals were:

To acquaint the participant with the Ada style of program development and software engineering methodology.

To enable the participant to write small-to-medium sized Ada modules and programs.

To instill a working knowledge of Ada's more advanced features including exception handling, generic units, and tasking.

To acquaint the participant with Ada coding style conventions.

To teach the use of Ada's modularity features in constructing software systems from reusable software components.

To emphasize the importance of software engineering practices through the experience of modifying code written by other participants and through programming in teams on larger projects.

PLANNING MODEL

The fundamental premise behind our planning for this Seminar was that college students and, therefore, college faculty need a different approach to the Ada programming language from that which is appropriate for working programmers. This premise is based on the following observations:

The vast majority of Ada training courses for industry are only a few days in length, and do not always include hands-on exercises. Essentially all the information must be presented by the instructor, with very little outside reading or assimilation time for the students. We believe that this leads to shallow learning of syntax and semantics, with little understanding of the theoretical basis for proper system design using the language. We believe this approach is not appropriate for teaching the language as a design tool.

College courses, on the other hand, emphasize individual study and research in conjunction with lecture presentations. The result is that college students are taught to apply the language as a tool for problem solving and to draw inferences from this activity as to what new applications may be developed. We believe that college faculty should be taught in the same way. We also believe that it is not reasonable for the average faculty member to attempt to learn Ada by self-study, as is possible with other languages. Ada is too complex and requires too much cultural assimilation for this to be effective.

We believe it is important in teaching Ada to college faculty to take advantage of the varied backgrounds of the participants. This can be done by relaying questions to members who may be able to answer them, by having members give presentations, by having them help one another with programming exercises, and other similar techniques.

We believe that one of the most important ideas to get across in teaching Ada is the concept of software maintenance and how Ada simplifies it. We intend to highlight this feature by requiring the participants to modify existing code under several different conditions.

CLASSROOM INSTRUCTIONAL ACTIVITIES

Classroom Materials

The primary classroom materials consisted of a series of overhead transparencies containing key points of information and many examples. These class notes have been developed over several years and tested in numerous courses. Students were given paper copies to facilitate note-taking and permit use as a reference. These were reduced in size, four overheads per page of class notes, to minimize duplication costs.

Texts

Two texts were used as primary references for the seminar:

An Introduction to Ada, 2nd Edition
S. J. Young, J. Wiley, 1984

Software Engineering with Ada, 2nd Edition
G. Booch, B. Cummings, 1985

Young, primarily a language text, provided an excellent supplement to the topics covered in class. However thorough the classroom explanations, few students are capable of assimilating all the details in one exposure. This material is absorbed more completely through a combination of classroom discussion, outside readings, and laboratory exercises. Young was considered very understandable by most of the participants; for those with weaker backgrounds, more elementary texts were recommended for the first reading. Booch, with its emphasis on software engineering principles and software design, was a valuable addition to the

regular readings. As early in the course as possible, students must have enough language features to solve meaningful laboratory exercises, but they must also understand the principles of software design with Ada in order to use Ada's features properly. These dual needs result in a competition for prime classroom time early, not only in this course, but in any course. Since students must have the language in order to write code, the language usually comes first. While the basics of software engineering and object-oriented design were included in later class periods, outside readings in Booch introduced these principles earlier than classroom time could permit.

Use of the Ada Language Reference Manual

The Ada LRM (ANSI-MIL-STD-1815A), the only completely reliable source of Ada information, is an essential student reference. Each participant was given a copy of the LRM at the first class meeting. It is of vital importance that the student become familiar with the LRM as soon as possible. However, to anyone not accustomed to using reference manuals, the Ada LRM can be both intimidating and confusing. Overheads made directly from the printed LRM page were used frequently in the class presentations to illustrate the use of particular features by means of the many excellent examples contained in the LRM. This use, in the context of the class, can go a long way toward reducing LRM-anxiety. Later on, when the students have a firmer foundation in the language, the LRM is regularly used to answer questions raised in class. [The class notes also make frequent reference to specific topics in the LRM.] Two primary reasons that the LRM is confusing are (1) precision in any language tends to increase the complexity and (2) many forward and backward references will inevitably touch on some features of which the student has little or no knowledge. The first difficulty is dealt with by increasing familiarity with the style and dissecting sentences, with the class, to analyze precisely what is being stated. The effect of the second gradually diminishes as the student learns more of the language so that fewer references are obscure.

Classroom Library

In addition to the texts and materials supplied to every student, a fairly extensive collection of reference materials was made available. These included numerous language texts, Ada reference books, periodical literature relating to current activities in Ada, and reference materials for VAX/VMS, VAX/Ada, and the EDT editor. Also in the library, as examples of available course materials, were (a) course materials used in the Ada training program at Keesler Air Force Base, (b) the courses L202 (Basic Ada Programming), L305 (Advanced Ada Topics), and L401 (Real Time Systems in Ada) developed as part of the US Army Model Ada Training Curriculum by SofTech, (c) materials for the tutorials Beginning Ada and Advanced Ada offered by ASEET (Ada and Software Engineering Education Team) and (d) overheads available for use with Booch's Software Engineering (1st Ed.).

Several of the reference books were thought to be important enough so that several copies were acquired for the library:

Rationale for the Design of the Ada Programming Language: In addition to being a good source of examples, this is an invaluable source of answers to such questions as, "Why were Ada loops designed to be so simple when tasks are so complicated?"

Ada as a Second Language. Norman Cohen, McGraw-Hill, 1986: This book, somewhere between an advanced text and a readable version of the reference manual, provided valuable information to the more advanced participants.

Software Components with Ada, Grady Booch, Benjamin Cummings, 1987: An affordable example of the software components industry predicted to develop as a result of Ada packages.

Order of Presentation of Topics

The most natural way to present a language as large and complex as Ada is top-down, with an overview of the history, philosophy, and structure of the language providing a context for the details. When the course involves laboratory exercises (unquestionably the best way to teach a language) and is presented as an intensive exposure with the end looming only a few weeks after the beginning, compromise must be made between the most logical order of presentation and the need to have details necessary for writing programs in the laboratory.

One compromise resulted in postponement of the history until late in the course; this was probably a mistake. The greater appreciation for the language through knowing its historical context, would be worth the small delay in writing more complex programs.

Another compromise, which was pedagogically sound and worked well, was to cover the language features in a natural order, but to treat them lightly the first time through and then go back again (and sometimes again) with more detail each time. The basic order of topics (with minor variations to accommodate specific needs for lab exercises) was:

- Overview of the language
- Program structure
- Discrete types (with necessary I/O)
- Statements (simple and compound)
- Procedures and functions
- Packages
- Input/Output (incl. files and formatting)
- Scope & Visibility (incl. Block statement)
- Separate compilation
- Object-oriented design & Software engineering
- Exceptions
- Generic units
- Access types
- Tasking

Class/Laboratory Daily Format

The schedule for class and lab periods was established early, and worked so effectively, for both students and instructors, that it was modified only for special circumstances such as guest lectures. This schedule was:

8:30 - 9:20 a.m.	Class
9:30 - 11:00	Laboratory
11:10 - 12:00	Class
12:00 - 1:00 p.m. Lunch	
1:00 - 2:30	Laboratory
2:40 - 3:30	Class
3:30 - 5:00	Laboratory

The laboratory was also available during evening hours, on weekends, and, by popular demand, early in the morning. The 50-minute class periods, interspersed with 1-1/2 hour lab periods seemed to optimize the learning experience.

Participants

Attending the Seminar were fourteen participants from fourteen institutions which ranged from 2-year colleges through universities and included a U.S. Army graduate school. Five of the participants possessed a doctoral degree, and all had advanced degrees in computer science. In an attempt to attain a relatively homogeneous group, the prerequisite of "experience in one or more modern programming languages, preferably including Pascal" was specified.

Every Ada course in which any of the seminar staff have been involved has had a heterogeneous mix of participant backgrounds; even with the stated prerequisite, this course was no exception.

The level of experience included those who had taught advanced Ada courses, those who had taught Computer Science courses but had only a cursory introduction to Ada, and those with substantial computer background to whom Ada was completely new.

A particularly gratifying response on all the end-of-course evaluations indicated that, in spite of the variance in preparedness among participants, "The seminar was very worthwhile for me." Apparently the complexity of Ada lends itself well to learning it on many levels. The beginners learned Ada concepts and structure without (in some cases) appreciating all the details, while the most experienced found that their prior knowledge of Ada was drawn together and the finer details were filled in. The variation was used to advantage in the lab assignments with stronger participants on teams with weaker ones; the weaker ones and the stronger ones learned from each other--on the one hand learning about programming in Ada, and on the other hand appreciating points of difficulty their students might have.

One surprising observation was that, in spite of the high level of experience of the class, the course syllabus was covered more slowly than in a typical undergraduate class. This was due in part because of the many excellent penetrating questions. As a result, the material was also studied in greater depth.

Guest Lecturer Program

Because of time constraints, only two guest lecturers were brought in. James E. Schell, formerly Director of the US Army's Center for Software Lifecycle Support, discussed the background and political issues surrounding Ada's development and use. Captain David A. Cook, US Air Force Academy, spoke on tasking. Both were considered by the participants to be valuable additions to the course.

LABORATORY ACTIVITIES

Overview

Ten programming exercises were given over the four week period. The programs were of ever increasing complexity and length and were coordinated with the material covered in lectures. Each new programming exercise required the programmer to use new features of the language. The assignments were scheduled to be assigned after the language feature had been covered in the lectures. All of the assignments were made in written form and placed electronically into each participant's directory separately and at appropriate times.

Even though all of the students were seasoned computer scientists most preferred the assignments in hardcopy form. This was evidenced by the fact that most students when given an assignment electronically immediately proceeded to get a hardcopy to read as opposed to reading it at the screen.

There were three team projects. On the first two team projects the team size was held to 2 or 3 per team. The final week-long project involved the use of separate procedures, generic and nongeneric packages, exception handling, and tasking. The team size was increased to 3 or 4 members for the final project.

The following is a list of the programming exercises showing the Ada feature that was stressed for each particular lab:

<u>Exercise</u>	<u>Title</u>	<u>Ada Feature</u>
1	First Ada Program	VAX/EDT/ACS
2	Sum of Integers	Program Structure, Text IO, Int_IO
3	Square Root	If, Loop, Function
4	Payroll	File Type, Procedure

<u>Exercise</u>	<u>Title</u>	<u>Ada Feature</u>
5	Math Library	NonGeneric Package, Exceptions
6	Income Tax	Array, Record, Nontext Files
7	Bingo, Team Project	Abstract Data Type, Package, I/O, Enumeration Types
8	Stack Package	Generic Package
9	Bridge, Team Project	Abstract Data Type, Package, I/O, Enumeration Types
10	TuAda Compiler Team Project	Tasking, Separate Compilation, Generic Packages

Team Projects

The first team project proved to be an interesting problem in that once the package was completed, the specification only was given to another team which was required to write a driver program to test the package. What was considered intuitively obvious to the package specification writer was not always crystal clear to the user of that package. The proper choice of function and procedure names and the parameters required by these subprograms proved to be a real challenge to the participants. This was the first attempt by the participants at object-oriented design and caused them to look at the problem in a different light. It was also their first use of enumeration types and enumeration I/O.

The second team project was similar to the first in that object-oriented design was required as well as enumeration I/O. The team members were different from the first project and an attempt was made to match more advanced participants with those less advanced. Things went more smoothly.

The major large project used larger teams because it was multiphase in nature and required a larger programming effort. The project was a miniature compiler project with three parts: a scanner, a parser, and a code generator. This compiler was different from most in that a buffer holding tokens was placed between the scanner and the parser and another buffer holding action routines and tokens was to be placed between the parser and the code generator.

The scanner was a table driven finite-state automaton; the students were given the table and the algorithm. It was simple enough but an understanding of its relationship to the symbol table proved to be a major problem. It was assumed that most computer scientists had rudimentary knowledge of compilers; this assumption proved to be false.

The parser was the central controlling unit and was an LL(1) table driven parser. The teams were given the parsing table; the production rules and the parsing algorithm were provided. The students' innate curiosity was greatly underestimated. Instead of treating the problem as just an exercise using parsing, they wanted to know more about compilers and not only how the parsing table was used but how it was generated.

The code generator was the easiest because syntax-directed translation was used and the names of the action routines were embedded directly into the production rules. The code generator had but a few action routines that generated quadruples.

COMPUTER SYSTEM SUPPORT

System Configuration

Hardware: Computer support for the seminar was provided by a VAX-11/780 which contained 16 Mbytes of memory, 968 Mbytes of disk storage, and 40 ports. A Digital LA-210 serial printer was spooled remotely for hardcopy output. The 16 VT-220 classroom terminals were connected to the computer across the campus by statistical multiplexers and a MICOM 600 Port Selector. This configuration proved adequate for this size program, even though the system was shared at times with up to 20 Fortran programming students and various researchers. A system of this size should be able to accommodate 20 to 30 Ada students if properly managed.

Software: The Digital Equipment Corporation VAX Ada compiler and the VMS operating system environment provided excellent support for the Seminar. The system was managed as recommended in the compiler installation instructions. Interactive users were given a Working Set (physical memory) allocation of 1500 pages (.75 Mbytes). A batch queue was established with 3500 pages (1.75 Mbytes) working set for compilations. Ada compilations, even with the DEC production quality compiler, require a great deal of memory, and will generate huge numbers of page faults (disk reads) if they are not given enough. Running compilations from a batch queue imposes a limit on the number of compilations running simultaneously, allows the jobs to use all the memory they need, and finish quickly. It also allows participants to be working on other tasks while the compilations are running. The result is efficient use of the system and minimal degradation.

System Organization

Faculty and participant accounts were placed in the same User Identification Code (UIC) Group to facilitate communications and file transfers. All assignments were made by broadcast transmission of files from faculty to user directories. Faculty had Group privilege, allowing them access to participant files for review and critique. Participants wishing to do so could set their default file protection to preclude other participants from

gaining access to their files. The grouping of directories was intended to be a technique to minimize printing requirements, but we found that even computer scientists are addicted to paper, and wanted to print out virtually everything they did, even though it was not really necessary.

SEMINAR LOGISTICS

Classroom/laboratory facilities

Lectures were presented in a classroom separate from the laboratory but nearby. The laboratory was one of the normal University student terminal labs, configured for up to 19 terminals. This arrangement proved satisfactory in all respects. There were no distracting terminal or system activities during lectures, and the movement between rooms provided a good break. Additionally, the use of two rooms allowed the seminar faculty to work to prepare lectures, lab exercises, etc. in whichever room was not in use. The only problem which arose was that some of the participants wanted to work long or unusual hours, and the control of the room key became a logistics exercise in itself.

Budget

The budget total of \$54,000 covered faculty salaries and expenses, course materials, computer operation and maintenance, seminar logistics, and overhead. Participants were not paid a stipend, but their on-campus lodging was paid. Funding support was provided through a U.S. Army grant.

Seminar Staff

Three people performed various major tasks in support of the seminar. The Academic Director was the primary instructor; thus, the primary selection criterion was lengthy teaching experience, including the teaching of Ada. The Lab Director developed and administered the programming exercises; thus, the primary qualifications were facility with the language and the ability to work well with people. The Systems and Logistics Director's job was to handle all system actions and logistics arrangements. The qualifications for this job were primarily managerial. Due to the critical importance of good computer support, it is essential that this person occupy a position of authority with respect to the computer system and the people who directly operate it. Other personnel involved were the regular computer services technical support and administrative staff of the host institution.

ANALYSIS OF PARTICIPANT RESPONSES

Participant reactions to the program were obtained through informal discussions during the seminar, during a scheduled verbal discussion session, and through comprehensive written assessments. Comments can be categorized as follows:

1. **Program length:** The four weeks was found to be adequate but ambitious for the material covered. Most participants would have preferred more time, but agreed that a longer program is a problem for most college faculty without other summer income.
2. **Program objectives:** There was considerable misunderstanding of the program objectives. Some participants were correctly expecting an entry level but comprehensive course comparable to the first one-semester graduate Ada course. Others expected an advanced program which began from a good foundation in the language. This perceived ambiguity in the stated objectives was said to be the cause of the diversity of experience levels, and resulted in both beginning and advanced participants gaining less than they had hoped from the program.
3. **Management of diverse backgrounds:** Several suggestions were made for taking better advantage of the advanced backgrounds of some participants. Such tactics as optional advanced lectures, more structured tutoring of the beginners by the advanced, having advanced people actually present topics from their experience were proposed. The best tactic, however, was believed to be separate, better defined programs for beginners and advanced. The consensus was that even greater benefits could be gained if the participants were of more comparable backgrounds Ada-wise.

CONCLUSIONS AND RECOMMENDATIONS

Program length and composition

Four weeks seems to be the best compromise between the divergent issues of time required to present an exhaustive treatment of the subject with adequate lab experience and time available to prospective attendees with other commitments and opportunities.

Contract timing

The availability of funds nine months before the seminar enabled recruitment of participants beginning in January, allowing wide dissemination of program information. Most participants applied in March, enabling confirmations to be issued in late April. Issuance of confirmations in March would be preferable, but it is difficult to get applications submitted in time to do that.

Academic considerations

The seminar was so successful that there are very few recommendations for change. One would be, as described above, to keep the history of Ada in the beginning of the seminar. While this might mean doing simple programming exercises for a little longer, until the Ada features necessary to more challenging programs are studied in class, this trade-off would be advisable. Another possible change would be to have a more homogeneous group of participants, if that is possible. Since the less well-prepared participants found the course to be valuable, it would be unfortunate to exclude them. While they would have received greater benefit had they been more familiar with Ada concepts, if not with the detail, their presence did not diminish the value to other participants and may well have increased it in some respects.

Laboratory considerations

The participants didn't get enough practice with tasking. Not all of the teams got their toy compiler to the stage where they could implement the buffer controlling tasks. One team that did get that far had some problems and decided to use the symbolic debugger. Everytime they used the debugger, the program worked fine, but it would not run without it. This presented a very interesting problem in that the asynchronous tasks did not work properly unless the debugger was inserted into the equation and it then changed the time so that the task worked properly. This experience really drove home the point that solving tasking problems is not as easy as debugging sequential code. Any added disturbance, the symbolic debugger in this case, can cause the timing of the asynchronous processes to be altered dramatically.

In retrospect, a project that required less of a learning curve and one that creates less curiosity could possibly have provided a more meaningful tasking experience. A tasking problem involving something that all of the participants were already familiar with might have enhanced the tasking part of the project. The Ada experience was valuable. The teamwork experience was valuable (teachers are accustomed to working alone). Dealing with software engineering aspects of a larger team project was also a very valuable experience.

Group size and composition

The group size of fourteen was good for program administration, but up to 25 could have been accommodated with the facilities and system support available. The group composition of all computer scientists was good in that instruction in elementary computer techniques was not required, but there was still a great diversity in backgrounds in the group. It is clear that more consistent expectations of the program are needed; it is much less clear how that can be achieved.

Budget considerations

The budget was adequate for the cost items which were included. It would be more advantageous to the program if sufficient funds could be made available to provide participant stipends as well. Most college faculty supplement their academic year income by obtaining teaching contracts or research fellowships in the summer. It is difficult for many to give up these summer activities in order to attend a seminar for which they are not paid.

Staff requirements

The staff of three professionals plus support from the normal computer services staff was adequate. Fewer than three would not allow the diversity of skills required for best success, and would impact staff ability to react to unexpected situations such as unusually diverse participant background. Dedicated clerical and system support is not required if the host staff are responsive to requests and problems. Most critical of the host support requirements is data communications if the lab is remote from the computer center.

Seminar results vs. planning model

Observed results and participant responses supported the validity of the premises set forth in the planning model. The approach of this seminar, with intensive instruction coupled with extensive laboratory experience was successful with this group. It was the consensus of participants and faculty that a shorter, less detailed seminar would not have provided the information required in order for the participants to become "Ada evangelists" at their home institutions.

Conclusion

This seminar was highly successful in virtually all respects. While the classroom and laboratory planning and execution were primary factors in this success, attention to participant convenience and comfort were also important to the overall learning environment in a residence situation. Credit must also be given to the participants themselves, who quickly became a very close-knit group and socialized together away from the seminar. The four weeks became a very personal and meaningful experience for all involved.

BIOGRAPHICAL DATA

Dr. M. Susan Richman is Director of the Ada Education and Software Development Center, and Associate Professor of Mathematics and Computer Science at The Pennsylvania State University at Harrisburg, Middletown, Pennsylvania. Dr. Richman is a graduate of the University of California, Berkeley, and received the Ph.D. in Mathematics from the University of Aberdeen, Scotland.

Dr. Charles G. Petersen is Associate Professor of Computer Science at Mississippi State University. Dr. Petersen is a graduate of Iowa State University and holds the M.S. degree in Computer Science and the Ph.D. degree in Higher Education from Iowa State University.

Mr. Donald C. Fuhr is Director of Computer Services at Tuskegee University, Alabama. He is a graduate of Oregon State University and received the M.S. degree in Engineering Management from the University of Alaska.

Training COBOL Programmers in Ada

by

Jagdish C. Agrawal
Computer Science Department
Embry Riddle Aeronautical University
Daytona Beach, FL 32014

Abstract: Multi-million lines of COBOL code has been in use in large organizations for a long time. With changing requirements, much of this code is ready for a major system change requirement, or for a redesign. At the same time, many large organizations with such code have changed their language standard from COBOL to Ada. However, these organizations and their support contractors have very valuable human resources of seasoned COBOL programmers who can be easily retrained.

There seems to be some controversy in Academic about how easy or difficult it may be to train a COBOL programmer in Ada. In this paper, the author is proposing a framework for an introductory course specially designed for the class of experienced COBOL programmers. In this approach, we are capitalizing on the previous knowledge of the student for the purpose of introducing new knowledge.

INTRODUCTION

The term "packaging" is familiar to system developers who have been using the Yourdon and Constantine's Structured Design techniques [1], which uses this term to refer to the assignment of modules of a total system into sections handled as distinct physical units for execution on a machine. COBOL programmers who have been practicing Structured Design techniques are very likely to find Ada's packaging capabilities. separation of package

specification and package body, and separate compilation very attractive and enjoyable. The prospects of gaining control on "visibility" and "scope" and thereby on abstraction, hiding, localization and modularity will make their learning of Ada exciting and commensurate with the software engineering practices they are familiar with already!

While in the literature there have been papers and books written on comparison between Pascal and Ada, there has been little work in the area of examining Ada, capitalizing on the knowledge of COBOL programmers. I believe that the job of training COBOL programmers will be made much easier if extensive literature existed on comparison and contrast between COBOL and Ada. This paper makes a small contribution towards such literature and it provides interesting similarities and dissimilarities of practicing software engineering with Ada. For example, the data structures, control structures, and

the module structures in the two languages will be compared and contrasted with examples.

In large organizations like the U.S. Army Information Systems Engineering Command (ISEC), where COBOL has been the programming language of choice for quite some time, now there is a need to train a very large number of seasoned COBOL programmers in Ada. However, at the outset, Ada appears to be intimidatingly large and complex compared to COBOL. Such perceptions contribute towards making the job of training experienced COBOL programmers in Ada a very difficult and challenging task. However, serious research to identify some strong similarities in the two languages and in the ways of practicing software engineering with the two languages can make the initial many lessons of a training program very easy for the COBOL programmers to understand. The author believes in the principle that initial success breeds more success. This in turn increases the productivity from the training program.

NEED FOR TRAINING EXPERIENCED AND PRODUCTIVE COBOL PROGRAMMERS

Requirements for redesigning large systems or developing new ones in environments that are adjusting to change in the implementation language standard from COBOL to Ada are expensive to implement and they involve a very large investment in "porting" the human resources from one environment to another. Seasoned programmers of the old environment with considerable experience with the outgoing environment find themselves under pressure and stress to expeditiously learn everything about the new environment and at the same time keep meeting the deadlines of the maintenance tasks on the outgoing environment.

FEASIBILITY OF TEACHING ADA TO COBOL PROGRAMMERS EFFICIENTLY

Designers and programmers in the COBOL environment have been using structured design techniques like those of Yourdon and Constantine [1]. These software developers have been practicing

modular design with emphasis on uncoupled or loosely coupled modules with functional cohesion (see, for example Yourdon/Conatantine [1, pages 84-140] as important criteria for the goodness of design. These concepts of coupling and cohesion are strongly linked to the software engineering principals of modularity, localization, hiding, and abstraction. Ada supports these principles very strongly. Naturally, seasoned programmers of COBOL can learn Ada and make this learning enjoyable by capitalizing on their experience with good design practices.

One can also achieve efficiency and productivity gains in the Ada training program for the class of seasoned COBOL programmers by capitalizing on the experience and understanding of these programmers with the data structures, control structures, module structures, and the good design principles practiced when they used COBOL.

Several organizations in the Department of Defense, where the use of Ada has been mandated, have many

programmers with experience in COBOL programming language. Capitalizing on this experience in the training program for Ada can be very useful in increasing the productivity of the training program, and also in reducing the front-end cost of the change in the language standard at these organizations.

EARLIER WORK IN TEACHING ADA AS A SECOND LANGUAGE

Arthur Jones et al [2] have investigated the feasibility of a CAI tool for transitioning COBOL programmers to Ada. This Ada CAI tool uses analogy to make learning more efficient. It attempts to explain Ada concepts and techniques in terms of COBOL analogues.

Agrawal and Hilburn [3] have proposed that "The benefits of the reverse engineering process can be realized if the redesign team has education in both the language in which the system was originally implemented, and in Ada, the language of choice for the implementation of the new redesign. Therefore, there is

merit in educating members of such redesign/implementation teams in Ada as a second language."

Because of the importance of Abstract Data Types, and Abstraction, it is important to emphasize these topics. The knowledge of COBOL Data Structures, Control Structures, and Module Structures can be used to introduce Ada standards on these topics. Wide acceptance of COBOL 85 (see, e.g., Jerome Garfunkel [10]) has increased the number of such Ada-COBOL parallel constructs on the above mentioned structures. Teaching modules that capitalize on these Ada-COBOL parallel constructs are likely to significantly increase the success rate of teaching Ada to COBOL programmers. This paper proposes a framework for a course to teach Ada to COBOL programmers capitalizing on the prior experience of COBOL programmers.

PREREQUISITES FOR TEACHING ADA TO COBOL PROGRAMMERS

Experienced COBOL programmers already have been practicing structured

programming techniques since the late seventies (see, e.g., Tyler Welburn [4]). Therefore, for the select class of people we have in mind (experienced COBOL programmers), we can presume knowledge of structured programming techniques and need to state this assumption as "a prerequisite for the proposed course. In fact, we need to state all the assumptions being made about the class of students who are expected to take this course:

1. Students should have reasonable experience of programming in COBOL, and also be familiar with the COBOL 85 standard [5].
2. Students should be familiar with modern program design techniques (structured coding, top-down design, stepwise refinement, etc.).
3. Students should have some experience with problem solving courses (e.g. math or science courses) or computer programming experience beyond a single course.

The framework for this course is being developed in this paper for the non-traditional student who is an experienced COBOL programmer and therefore easily satisfies the above prerequisites, or can acquire them quickly by self reading or through a guided study of these prerequisites.

OBJECTIVES OF THE PROPOSED COURSE

The learning objectives of the course are:

- a. Study and practice of the basic features of the Ada programming language; using comparison and contrast with COBOL. Educators will need to develop detailed tables of comparison and contrast of the features of Ada versus those of COBOL 85. Some examples appear in the following sections of this paper.
- b. Use of packages and their support for modularity and localization, and their support for the development of reusable, easy to maintain program units;
- c. The various Ada features that support abstraction and information hiding.

TOPICAL OUTLINE

In the topical outline below, frequent reference has been made to the Language Reference Manual (LRM) or the Reference Manual for the Ada Programming Language.

1. INTRODUCTION (6 hours)

- a. history of the development of Ada
- b. structured program design and pseudocode programming
- c. software engineering and the principal features of Ada
- d. commonality of goals of structured program design and software engineering and early exposure of the Ada features that support these goals.

2. BASIC ADA CONCEPTS (10 hours)

- a. Predefined types -- BOOLEAN, CHARACTER, INTEGER, ENUMERATION, BOOLEAN, and REAL. (LRM Chapter 3)
- b. context clauses and instantiation of generic packages in text_io, and exposure to the package standard (LRM 7.1, 8.4, 10.1, 10.1.1)
- c. structure of an Ada program declaration part and execution part declarative region, scope of declaration, visibility, executable statements (LRM 3.1, 8.1, 8.2, 8.3, and 5.1)
- d. input/output (LRM 14.3, 14.3.1 - 14.3.10, and 14.4)
- e. an Ada compiler and its environment (LRM 10.1, 10.4, 10.5)

3. OPERATORS AND CONTROL STRUCTURES IN ADA VERSUS COBOL 85 (6 hours)

a. expressions (LRM 4.4)

b. control structures - if, case, loop, goto (LRM chapter 5)

4. DATA STRUCTURES (8 hours)

a. attributes, subtypes, derived types, conversion between types (LRM 4.1.4, 3.3.1, 3.3.2, 3.3.3, 3.4, and 4.6)

b. enumerated data types (LRM 3.5.1, 3.3, 3.3.1, 3.5, 3.5.1)

c. arrays and strings (LRM 3.3, 3.6, 3.7.2, 4.6, 3.6.3, 4.2)

d. records (LRM 3.3, 3.3.1, 3.7, 3.7.4)

e. files (LRM chapter 14)

5. SUBPROGRAMS IN ADA VERSUS COBOL 85 (6 hours)

a. functions and procedures (LRM 6.1, 6.4, 6.5)

b. operators and overloading (LRM 4.4, 4.5, 8.3, 8.7)

c. recursion (LRM 6.1, 6.3.2, 12.1)

6. PACKAGES (5 hours)

a. specification and body (LRM 7.2, 7.3)

b. visibility and scope (LRM 8.2, 8.3)

c. library units and order of compilation (LRM chapter 10)

d. generic units (LRM chapter 12)

e. Ada compilation units, library units, order of compilation, program library, and elaboration of library units (LRM 10.1, 10.1.2, 10.2, 10.2.1, 10.3, 10.4, 10.5)

7. ADVANCED TYPES (3 hours)

a. access types (LRM 3.3, 3.8)

b. variant parts (LRM 3.7.3, 4.1.3)

c. private types (LRM 3.3, 7.4)

8. TASKS (5 hours) (LRM chapter 9)

a. real time applications in data processing and concurrent programming

b. tasks and rendezvous

c. task types

10. MISCELLANEOUS ADA TOPICS (3 hours)

a. exception handling (LRM chapter 11)

b. implementation dependent features (LRM chapter 13, and vendor supplied LRM's Appendix F)

c. support available in the Ada community

d. trends and conclusions

USEFUL COMPARISONS FOR THE COURSE

CONTROL STRUCTURES IN COBOL 85 AND ADA

1. If Structures

```
Ada:      if <condition> then
           <s_o_s>*
         { elsif <condition> then
           <s_o_s> }
         { else
           <s_o_s> }
         end if;
```

```
COBOL 85: IF <condition>
           THEN <s_o_s>
         { ELSE IF <condition>
           THEN <s_o_s> }
         { ELSE <s_o_s> }
         END-IF.
```

* s_o_s => sequence_of _statements

MODULE STRUCTURES IN COBOL 85 and ADA

System maintenance is a natural event. Accepting this basic fact, It is important to design systems with the maintenance function in mind. This philosophy has been a driving force in establishing modifiability and understandability as two of the four goals of software engineering for the practice of all programmers as proposed by Ross, Goodenough and Irvine [7] and Booch [8].

Poorly structured and poorly documented systems developed in COBOL in the sixties or the early seventies are fast reaching their obsolescence. Many of these obsolete systems are so out of control that they have to be discarded and replaced with completely new systems.

Fundamental theorem of software engineering proposed by Yourdon and Constantine [1]:

$$C(P/2) + C(P/2) < C(P)$$

points out that the complexity of a problem can be reduced by dividing the problem into several smaller problems. Decomposition of a function into component

subfunctions is the tool for achieving the software engineering goal of modularity [7]. COBOL practitioners have been practicing it for some time now and perhaps would welcome many features in Ada that are not available in COBOL 85, but these features enforce modularity principle.

Examples of modules include procedures, subroutines, and functions. Modularization allows the designer to decompose a system into functional units, and impose hierarchical ordering on function usage, as well as implement data abstraction, which Ada allows more than COBOL 85. An important software design objective is to structure the software product so that the number and complexity of interconnections between modules is minimized. Special rules on scope and visibility within and outside Ada packages give the designer a strong control on the interconnections. Separate compilation of packages and package specification and package body also give a strong control for minimizing interconnections.

REFERENCES

1. Ed Yourdon and Larry L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice Hall, Inc., Englewood Cliffs, NJ, 1979.
2. Arthur Jones, Daniel Hocking, and Jagdish Agrawal, "A Tool for Transitioning COBOL Programmers to Ada," Empirical Foundations of Information and Software Science IX: Empirical Methods of Evaluation of Man-Machine Interfaces, editors: Pranas Zunde and Jagdish Agrawal, Plenum Press, New York, 1987. pages 415-422.
3. Agrawal, Jagdish and Hilburn, Thomas, "Ada as a Second Programming Language," Conference Proceedings of Ada Expo '88, October 9-12, 1988, Ada Expo, Anaheim, California, 1988, Section X: Ada in Training and Education.
4. Welburn, Tyler, Advanced Structured COBOL, Mayfield Publishing Company, Palo Alto, California, 1983.
5. Gerome Garfunkel, The COBOL 85 Example Book, John Wiley & Sons, New York, 1987.
6. Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A-1983, United States Department of Defense, Washington, D.C., 1983.
7. Ross, D. T., Goodenough, J. B., and Irvine, C. A., "Software Engineering: Process, Principles, and Goals," IEEE Computer, May 1975.
8. Booch, G., Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

TEACHING ADA FROM THE OUTSIDE-IN

Donald P. Purdy

Manatee Community College, Bradenton, Florida

Abstract

Teaching Ada from the outside-in is a concept which the author conceived after attending the 1988, Department of Defense, Ada Curriculum Development Seminar held at Tuskegee University, Alabama. The seminar members agreed that the more difficult topics of exception handling, tasking, real time interfacing, user defined generics, access types and discriminant records could be ignored in a first year Computer Science course (CS-1). The author agreed with these exclusions, but felt that without these items, the true power of Ada could not be taught. The outside-in concept was designed to teach the basics and the advanced subjects in a way that would maintain student interest while they worked on the more mundane aspects of the language. It would also lead the students into using the advanced tools that Ada provides. This concept was used in our first Introduction To Ada class in the Fall of 1988.

Methodology

The instructional plan included:

1. Developing a schedule/syllabus that would allow teaching simple and complex subjects at the same time.
2. Identifying advanced students and neophytes using an experience survey form.
3. Scheduling students for different class hours and different subjects depending upon their experience.
4. Using videotapes for the simple topics with instructor controlled, tape playback.
5. Using an overhead and detailed program walk-thru for the advanced subjects.
6. Getting the students interest and involvement in the outside-in concept during the first class meeting.
7. Maintaining that interest by showing beneficial usage of the advanced subjects.

8. Relating advanced subjects to scheduled course topics whenever possible.

9. Developing follow up, advanced subject presentations, by expanding or refining the applications previously used.

10. Maintaining an active interest in student programming efforts.

Student Data

The student survey forms produced the following student data:

Math codes indicate:

0 - H.S. Algebra 3 - Clg Calc I
1 - Clg Alg I 4 - Clg Calc II
2 - Clg Alg II 5 - Clg Calc III

Stu ID#	Wkg Hrs	Ada	Pascl	COBL	C	FORT	BAS	Math
1	52	x				x		3
2	52	x	x		x			3
3	65		x				x	2
4	56		x		x	x	x	2
5	45		x	x		x	x	3
6	36		x		x	x	x	5
7	69		x				x	1
8	52		x		x			1
9	60		x				x	2
10	52		x			x		2
11	52		x				x	3
12	52		x			x		3
13	62					x	x	0
14	52					x	x	3
15	52					x	x	3
16	48		x	x		x	x	3
Totals		2	12	2	4	9	11	
Corrected		0	7	2	2	9	11	

Further detailed questions in class led to the corrected data line. The two students, who had indicated Ada experience, had written short, 20 line programs. The Pascal and C numbers were revised for a like reason. The Wkg Hrs column figures are based on a combination of actual working hours plus an estimate of 12 hours of study per week for each 3 semester hour course. This information and the math data are used to identify students who may

have difficulty completing the course with a satisfactory grade. This class had a very high drop/fail potential considering the working hours and a very low potential considering the math data.

The corrected data indicate an almost equal level of structured language experience and non-structured experience. This class make-up was almost perfect for using the outside-in approach. The students having structured experience were our own Computer Science majors who chose the course as an advanced elective. One of these was a quadriplegic with almost total motor and speech impairment. Most of the remaining students worked for a local defense contractor. These students were very difficult to guide, and were needlessly disruptive during the initial classes. One student, self-taught in BASIC and Pascal, worked for a local Farm Credit Bureau. To later identify a student or a class of students the following codes will be used:

- S - Structured experience
(Pascal, Modula-2 & C)
- N - Non-structured experience
(BASIC & FORTRAN)
- L - Low math experience
(No college math)
- H - High hour schedule
(Over 60 hours)

The Environment

Eight of the students did their work on a 1983 validated compiler running on a VAX 11/750. This compiler did support generics. Integer, Float, and Boolean I/O were handled by packages nested in the Text IO package. During the latter part of the semester, the company these students worked for, locked out the student Ada passwords until 5 PM. It seems the Ada compile runs were deteriorating the computer production capacity during daily backups. The rest of the class completed their work using a popular microcomputer compiler running on the college 640K, hard drive machines.

The Schedule

Wk Regular No Subject	Advanced Subject
1 Introduction	
2 Names In Ada Constants & Expressions	Using Ada for Design Instantiation
3 Pre-defined Types (Class 3 was cancelled due to weather)	Low_Level_IO
4 Pre-defined Types User Defined Types	Low_Level_IO Exceptions
5 Control Structures	Tasks & Packages

Schedule (Continued)

Wk Regular No Subject	Advanced Subject
6 Arrays/Test #1	Tasks & OOD ¹
7 Strings & Records	Task Types
8 Subprograms	Generic Subprograms Recursive Functions
9 Generic Subprograms Packages	Generic FIFO Pkg ²
10 Formatted I/O Overloaded Operators	Discriminants Private Types
11 Discriminants/Test#2	Access Types/Files
12 List Processing Private Types	
13 Exceptions and Files	
14 Tasking	
(14 Cancelled due to Tropical Storm Keith)	
15 Tasking, Low Level IO and Using Ada for Design	
16 Review & Final Exam	

The schedule was altered by two weather systems we experienced during the semester. This caused doubling up on some class content, but at the same time allowed the instructor more time to prepare concise, direct, more meaningful presentations. Wherever possible, the advanced subjects were presented so as to improve upon the regular subjects.

Assignments

Assignments, two per chapter through the twelfth chapter³, were contained in the course syllabus. This left 5 weeks at the end of the course for work on the team projects. The students set up their own teams with the instructors approval. Approval was based on each team having at least one member with structured programming experience. Each team was allowed to design their own final project although they were strongly encouraged to develop a cruise missile autopilot program. Most teams opted for other final projects, claiming insufficient knowledge of aerodynamics or flight control systems.

Teacher/Student/Subject Interaction

During the first two classes, the students were asked to confirm their acceptance of and participation in the outside-in concept. The response was positive each time. The third class was canceled due to severe rainfall and local flooding. The advanced subject for the fourth class was real time interfacing, a difficult item, especially since the compiler⁴ did not support the subject. As the teacher began the presentation, the students revolted. They had not prepared for the subject. Learning the syntax and style of Ada plus the complex features was too much. At this point, the students and

the teacher reached a compromise. The presentation of advanced subjects would continue, but the students would only be required to study the regularly scheduled subjects.

The class continued with real-time interfaces presentation followed by the regularly scheduled subject and then another advanced subject, exception handling. Students were advised that they did not have to stay for advanced subject presentations for this class or any other unless they desired. It is interesting that no one skipped any of the presentations on advanced subjects even though they were given the opportunity. Fear of being left behind may have been the cause. Student involvement was the very high during coverage of advanced subjects.

Tasking was the advanced subject for the fifth. Using an overhead projector to display the programs and the output (text and float), a walk-thru showed the program progression mixed with task actions. The remaining classes were conducted using video tapes for the simple topics and an overhead walk-thru for the advanced.

In-Class Participation

Student participation ran from disbelief (exception handling) to appreciative awe (tasking and generics). The use of packages in program development brought high interest. Controlled playback of the instructional tapes allowed more timely and pertinent student questions. The overhead presentations of advanced topics evoked high student involvement. The students could proceed, arguing and discussing the subject, with the instructor joining in at points where clarification or an opinion was required. This usually fostered further discussion.

Drop Rate and Causes

A 12.5% drop rate was recorded when two of the sixteen students withdrew from the class. One of these students was forced to miss several early classes due to surgery/recuperation. The second student (#13-N,L,H) had a very high drop potential based on his student evaluation form. His dropping proved the validity of the form. Three students were questionable until the fifth class when they opted to stay on board. These were all N types.

Problems Encountered

The first problem with teaching Ada is the general Ada programming mind-

set. That is, develop useful programs that do not depend on or support any other programs. The acceptance of reusable packages and multi-use generics is a small hurdle to overcome. The second problem, inbred by BASIC and FORTRAN, is the tendency to be terse with identifier names. The third problem is a manifestation of the second. Presentation of complicated, interrelated, programming concepts calls for judicious use of media space. This may lead to bad examples of self-documenting identifiers or the use of simplistic, useless, program examples. The first problem can be overcome. The second and third will require some work. A problem relevant to this class was the use of two different compilers. The older version had fully defined I/O packages for Integer, Float, and Boolean types. The newer model required instantiation of generic I/O packages for these types.

Conclusion

The outside-in approach obtained good results using early introduction of, and student involvement in, the advanced and more interesting aspects of Ada. Early discussion of advanced topics made later coverage more meaningful. When the topic was revisited, student participation was more intense resulting in greater understanding of the subject. The concept worked well with this particular group; however, testing is needed on a CS-1 class where most of the students are first year Computer Science majors.

References

1. Grady Booch, Software Engineering with Ada, Addison-Wesley, 1981, Chapter 7.
2. S. J. Young, An Introduction to Ada, John Wiley, 1983, pages 273 - 274.
3. Sabina Saib, ADA: An Introduction, Holt, Rinehart and Winston, 1985.
4. JANUS/Ada User Manual, R & R Software, pages 13-5 and 14-38.



Don Purdy is a Computer Lab Supervisor at Manatee Community College. The lab supports Ada, Assembler, BASIC, C, COBOL, FORTRAN, Pascal, RPG, and application software for CAD/CAM, data base, word processing, and spread sheets. The lab hardware consists of 46 PC-XTs, 10 IBM System 38 terminals and 7 VAX 11/750 terminals. A retired USAF pilot, Mr. Purdy graduated Cum Laude from the Computer Science Program at Manatee Community College.

An Intermediate - Level Problem Set For Experienced
Programmers Or Writing Ada Code That Achieves
The Language Goals

Ruth S. Rudolph

Computer Sciences Corporation
Moorestown, New Jersey 08057

ABSTRACT

The DoD expected the Ada language to be a solution to spiraling software costs. However, the development of a tool and the proper use of the tool are not the same. DoD understood that realizing these cost savings would depend (among other things) on rigorous training in the correct use of the tool. Improper or inadequate training in this language tool will produce code which accomplishes none of its goals and costs more to write. From experience it seems that the longer a person has been in the industry, the longer it will take for him to learn this new language because previous languages required the problem to be implemented from the hardware perspective. Since Ada views the problem differently, retraining of experienced personnel is needed. A solution to the dilemma can be found in the student exercises which support the course work. This means that the problem set must be designed for experienced personnel but could be used for others.

INTRODUCTION

Experienced programmers have responded to the proliferation of programming languages developed over the last 30 years with minimal difficulty. Instruction in these languages has been a syntactic presentation given in a relatively brief time period, and the learning curve has been excellent. This is not the case with Ada. The time to learn to use this language tool effectively seems to vary in direct proportion to the years of programming experience. This presents a tremendous dilemma to management who find the increase in training costs difficult to accept and to the student who is accustomed to learning new languages with relative ease. It is essential to develop a curriculum that solves this problem because without successful adaptation to the new language, the goals for which it was designed will never be realized. In particular the costs of implementing in this language will be more rather than less.

OBJECTIVES OF THE COURSE

The Ada language is large and contains many difficult and subtle features. Subsetting of the language is not an acceptable approach. In fact the following issues must be addressed:

1. The student must not only be exposed to the entire language as specified in MIL-STD-1815A but must also develop expertise in which he can select appropriate features of the language for implementation in different circumstances.
2. A rigorous style convention must be adapted or the wordiness of the tool will discourage the coder.
3. Available tools (language sensitive editors and debuggers) must be used by the student.

None of this is possible unless a firm foundation has been laid prior to hands-on exposure which addresses the software engineering principles that are realized in Ada by using the following:

1. Abstraction
2. Information hiding
3. Data encapsulation
4. Reusable components
5. Fault_tolerant processing

SELECTION OF A PROBLEM SET

The development of a problem set (not a case study) which will:

1. Allow the student to utilize many Ada features
2. Demonstrate to the student the appropriateness of each feature

will help the retraining effort to achieve an effective technology transfer. Each problem in the set should include a template that can be copied and a set of instructions explaining the requirements. These should focus on very specific aspects of the language including:

1. Text_IO versus File_IO
2. Use Clauses
3. Attributes
4. Records with Discriminants
5. Derived and Subtypes
6. Abstract Data Types
7. Packages-Specification and Body
8. Access Types
9. Input/Output and Implementation Dependence
10. Use of tools
11. Private types

Individual problems are chosen instead of a case study because system experience gained from a case study approach is not the objective here. The objective is to learn to use Ada language features correctly.

PREREQUISITES

At the beginning of this course the student has received a foundation in the Ada language and has completed a minimal set of Ada problems and is able to

1. Create an Ada program library
2. Write a procedure, function and package private types
3. Use attributes, user defined types and private types
4. Combine program units by way of a context clause
5. Instantiate generics to do integer and enumeration Input/Output
6. Compile, link and run an Ada program

Now the students is ready to become an Ada programmer.

In the following exercises it is assumed that the VAX Ada Compiler and the VAX Ada program library manager(ASC) are used, but any Ada development environment can be used.

INTERACTIVE PROCESSING AND IMPLEMENTATION DEPENDENCIES

Input/Output is so difficult in most languages that usually only a few people know how it really works. In Ada, a reusable package has been provided and everyone can write I/O. However, the student must overcome his natural fear of this subject and also learn about certain implementation specific problems which must be overcome to achieve reusability. This topic provides an excellent opportunity to learn about reusability, its advantages and limitations, by writing I/O code.

In the preliminary problem set, the student has initialized all of the variables within the program so that he would not have to be confused with Input/Output. He has been required to write only Text_IO 'Put' statements. These were needed to demonstrate the correctness of the programs. The first problem in this course requires that one of the preliminary problems be rewritten so that data can be entered interactively from the terminal. The problem asks for a count of the number of non-blank characters in any string entered from the terminal. The solution is facilitated by the use of Text_IO.Get_Line. Some insight is needed to implement this:

1. A string object must be initialized to all blanks.
2. A constraint for the length of the string is required. (This limits the length of the string entered from the terminal. The program should inform the user of this with a prompt.)

Since the parameter list for Text_IO.Get_Line is defined as
(Item: out String; Last: out Natural);

Several limitations appear:

1. If the actual parameter which matches LAST is of type Positive (a subtype of Natural), an accidental depression of the carriage return at the terminal will cause a constraint error because a value of zero will be returned for LAST and that value does not exist for Positive(range constraints).
2. If the user wants to constrain the string to be very long and does this by defining the string as 1..Positive'Last(in agreement with the LRM) a numeric/constraint error will be raised because the size of strings on Dec's

Vax are limited to 65,000 (implementation dependency).

3. If the size of the string is constrained by a constant, that constant must be of type Positive or some subtype thereof to match the string constraint in package standard. It must not be a derived type (strong typing).

Problem 2 continues with more opportunities to experience input/output and an introduction to the 'use' clause. The student prefers to use the 'use' clause because it reduces the amount of code that must be written. Ada is a very wordy language and this is annoying to the new user. Since the student has been told that it is not good style to use the 'use' clause he must have an opportunity to understand the confusion that can result from its use and misuse. Once he has successfully implemented the problem without the 'use' clause a change is made which results in an error condition which can be solved with the 'use' clause. The complexity of the language is such that many aspects of a feature need to be demonstrated. As with some of the other problems, additional Ada features are utilized in the solution.

This is an inventory maintenance example where quantities of liquor are stored in a sequential file. The information for each 'record' includes the type, manufacturer, quantity, unit price, and value. These data values are entered interactively from the terminal and upon a request from the user the program will produce a total inventory value broken down by item.

The student is exposed to the following Ada concepts:

1. Text_IO
2. Sequential_IO
3. Records
4. Instantiation of generic units
5. Mixed types
6. Use clause
7. Exceptions

In addition the student must mail the teacher the inventory file in readable form. The student must read data of different types from the terminal, store the data in a sequential file, make calculations on the data in the file and translate the sequential file back to an ASCII file.

Furthermore the IO Exceptions can raise many errors so exception handlers must be written. One of the errors that becomes apparent is that strings are case sensitive and fixed length. The user must know to blank pad the string that is entered to match the

constraint in the definition. Also if you inquire from the user as to whether the session should be terminated, YES or NO and the response is No, an error will occur for two reasons, the length of the string doesn't match and the case doesn't match.

Other things that will be learned about I/O include the difference between Open and Create and the use of file mode. Certain implementation dependencies such as when the end of line terminator is read may also be discovered.

In addition, this problem is designed so that students with different capabilities can be challenged. A simple solution is acceptable but a more elegant solution will appeal to the more knowledgeable student.

When this problem has been submitted correctly, a change in the specification is made. Now all the data types for the records must be encapsulated into a package and then the problem is to be recompiled, relinked, and rerun. If any of the types in the package were derived types (and at least one of them should be since some of the types are of type Float from Package Standard and it is recommended to make a derived type of Float) an error occurs because of the hiding of the operators in the new package by the operators that are visible in package Standard in the main procedure (see N.Cohen-"Ada As A Second Language")⁴. Without the use clause, the solution to do this is very awkward. The student has a choice of:

1. Using selected component notation for an operator
2. Renaming the operator function "" (L.R: Liquor_Data_Package.Quantity, Liquor_Data_Package.Price) return Liquor_Data_Package.Inv_Value renames Liquor_Data_Package."";
3. Inserting the 'use' clause for the Liquor_Data_Package and clearly commenting why it has been added. In addition all the expanded names which have been included when the 'use' clause was not there should remain in place. A final touch to this problem to guarantee the use of a style convention is to submit the source to the Ada repository "pretty printer" program. The student must be sure the result is copied to a new file in case the changes made by the pretty printer are not acceptable.

A final touch to this problem to guarantee the use of a style convention is to submit the source to the Ada repository "pretty printer" program. The student

must be sure the result is copied to a new file in case the changes made by the pretty printer are not acceptable.

USING THE DEBUGGER AND DATA STRUCTURES WITH COMPOSITE TYPES

The next two problems are to be compiled, linked and run under Debug. The student has received instruction on how to use the debugger and has completed a trivial exercise which allowed him to practice with the debugger.

Ada has an excellent capability for modelling data with composite types. However, the syntax to implement these structures can be rather confusing. Even the experienced programmer finds these structures difficult to implement. These problems are included to force the student to practice with these constructs and become comfortable with them.

The two problems:

1. Building an atlas using nested records and arrays
2. Building a variable record using nested records with discriminants

were designed to provide experience in how to implement these structures so that in a real-world situation the student will not be afraid, for lack of understanding, to take advantage of this rich expressive capability.

Problem 3 for nested records and arrays requires five type definitions to create an array of records where one of the record components is an array of arrays and the inner array contains records. This is not a trivial exercise. The difficulty is in accessing some of the deeply nested components. Running the program under the debugger makes this easier to understand. However, the student now discovers some limitations in the VAX debugger as it supports the Ada language. Arrays of records with four fields are not supported. If an 'examine' is done of such a structure it will be found that the third and fourth fields receive the same default value for all of the aggregates even though an Ada 'Put' statement (using Text_IO) shows the aggregate values that are expected. No warning or error message is given so the novice may think an error exists in the program (using the debugger) when in fact no error is there.

In addition, when defining deeply nested composite types it is easy to raise a storage error. The student

should try each type before embedding it to be sure it is implementable.

Problem 4 on discriminants requires a description of boats where boats can be of two classes, sailboats or motorboats, and sailboats contain two subclasses. All of the boats have some components in common. The first class of boats also has some new unique components. In the second class, both subclasses have some components in common but also have some unique components.

There are three solutions to this problem. The correct solution is the one in which a component of the record with the discriminant is itself a record with a discriminant. The outer most record contains a discriminant that shapes the record as a motorboat or a sailboat. The components that are common to all boats are listed and then the variant part describes the components which are unique to sailboats and motorboats. For a sailboat the component names another variant part which allows the shape to model a schooner or a ketch. The other two solutions:

1. Creating two discriminants
2. Renaming equivalent components

do not result in very well defined structures. Creation of two discriminants requires a null value for one of the three kinds of boats which does not use the second discriminant and renaming equivalent components is not good modelling. This problem demonstrates to the student the difficulty of using this feature without abusing some of the languages objectives (i.e. readability). It also shows the student an excellent modeling technique-nested variant records-which may not have been intuitive.

DEVELOPING ABSTRACT DATA TYPES AND CONSIDERING ADT'S AS CANDIDATES FOR GENERIC PROGRAM UNITS

An abstract data type (ADT) can be defined as a data type with a set of operations valid for that type. Abstract data types involve the following three concepts:

1. Abstraction-extraction and presentation of the essential properties of a data type
2. Information Hiding-making inaccessible all implementation details of a data type
3. Encapsulation-grouping together the various details of a data type abstraction and its implementation

These concepts are supported in Ada by the following features:

1. Private types
2. Limited private types
3. Packages with private parts

ADT's are naturally implemented in Ada and help to enforce many of the language goals such as readability, reusability, and maintainability. The experienced programmer probably has never worked with ADT's since he has probably never built user defined types. The use of them is not intuitive and if the Ada language is to be used to its full advantage, it is essential that the student understand what they are and how to implement them. Problem 5 addresses all these things by building an ADT for sets and using the ADT to solve a problem which requires the generation of 4 sets. The set universe which is to be implemented is the universe of integers from 1 to 100. The application requires the computation of the following four sets:

1. Set divisible by 2,3,or 5
2. Set divisible by 2 or 3 but not 5
3. Set divisible by 3 and 5
4. Set not divisible by 3

The solution for this application requires the creation of an ADT where the universe is defined to be an array of Booleans and the array is indexed by a subset of integers constrained by 1 to 100. This array should be of type private so that the abstraction will be enforced and therefore it must be encapsulated in a package with all of the set operations. The implementation of those operations are hidden in the package body. It is useful when implementing an ADT for sets to declare two constants, one for the universe and one for the empty set. Since the type which models the universe is private, it will be necessary for the student to use deferred constants. It will probably be the first time the student has had the need to use them and therefore it is the first time he will understand their need and syntax. Deferred type definitions will be needed when access types are introduced so this is an important idea to develop. Not all of the possible set operations are required to solve this problem but they should all be included in the package because it becomes a candidate for a reusable component. In fact when the problem has been successfully completed the student should go back and rewrite the package as a generic. The formal type parameter should be defined as ($< >$), that is it should be capable of being matched by any discrete type because that allows the reusability to extend to a universe that includes things other than just numbers. The universe of discrete objects could consist of the alphabet or any lists of enumeration

literals such as train stations or other things you may want to identify as belonging to a set.

This problem is not particularly difficult to implement but it does provide an opportunity to apply the Ada language in a way which was intended by its authors.

SUMMARY

Even without formal training the experienced programmer will no doubt eventually learn to write Ada programs that will work but they will probably not achieve the language goals and will assuredly result in increased development costs. The major abuses that appear will be:

1. Lack of readability
2. Lack of reusability
3. Misuse of Ada features
4. Failure to use Ada features which contribute to the accomplishment of the language goals

This problem set was designed to help the student overcome these difficulties. The use of a uniform template as a starting point for the class helps to address the readability and reusability issues. The student is started in the right direction. The selection of these five problems highlights many of the important language features which are essential to writing good Ada code.

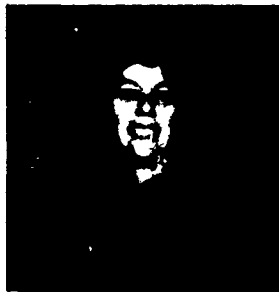
Those features include:

1. Input-Output (Sequential and Text)
2. Packages
3. Records
4. Functions
5. Exceptions
6. Private Types
7. Attributes

In addition, implementation dependencies and style issues are highlighted. Furthermore, the use of Ada development tools is required as part of the problem solution. This kind of direction enables the student to produce solutions that use good Ada code. The student gains an appreciation of what is expected from the language. When good coding habits are established during the initial learning phase, the results will be a continued growth in ability to use rather than misuse Ada. The DoD goals of reduced costs to produce code that is readable, maintainable, reliable, and reusable will be achieved.

REFERENCES

1. ADA83 - Reference Manual for the Ada Programming Language, ANSI/Military Standard MIL-STD-1815A, United States Department of Defense, January 1983.
2. ADV86 - Advanced Ada Topics (L305) Course Notes, U.S. Army Communications Electronics Command (CECOM), Ft. Monmouth, NJ 1986.
3. ADV84 - Advanced Ada Workbook, U.S. Army Communications Electronics Command (CECOM), Ft. Monmouth, NJ 1984.
4. COH86 - Norman H. Cohen, Ada as a Second Language, McGraw-Hill, Inc. 1986.
5. FEL85 - Michael B. Feldman, Data Structures With Ada, Reston Publishing Company, Inc., 1985.



ABOUT THE AUTHOR

Ruth S. Rudolph, Training Coordinator for the Tactical Systems Center at Computer Sciences Corporation, Defense Systems Division (DSD), Moorestown, New Jersey, is responsible for developing the internal technical training courses given within the center. Included in this technical training is an Ada curriculum, which she designed. For the last 7 years Ms. Rudolph has taught Ada courses throughout DSD.

A 10-DAY ADA COURSE FOR THE INDUSTRY

Perideon Mainian

Cameron University
Lawton, Oklahoma

Abstract

Teaching Ada is considered a challenging task for most educators. The size of the language, the presence of nontraditional structures and the complexity of parallel processing are some of the reasons for this challenge. Ada training in the industry involves additional constraints and brings new challenges to the instructor. This paper presents the experiences of one instructor in teaching Ada to industry professionals. The paper concludes with a collection of guidelines that have proven effective in conducting similar intensive courses.

Introduction

When teaching in academia, the instructor can usually make valid assumptions about the skills of the students. These assumptions generally provide guidelines to selecting the proper level of instruction and the amount of coverage; however, these assumptions can not be applied when training industry personnel. The training period is usually much shorter and the students come from diverse backgrounds.

The students who attended the 10-day Ada course are managers, software engineers, programmers and other personnel. Some have limited programming skills while others may have experience in FORTRAN, COBOL, Assembly or a military language. Unlike academia, only a few of the students have familiarity with Pascal. This diversity creates a new difficulty in teaching Ada in such a short period.

Student Goals

Each student has a different reason and goal for attending the course. Managers are interested in becoming familiar with basic principles and features of Ada. Software engineers may be interested in learning how Ada can help them with design issues in complex projects. Programmers simply want to learn all of the language details since they will be coding in Ada in the near future. As difficult as it may seem, it is possible to incorporate a careful and non-aggressive approach in presentations that can satisfy the expectations of all attendees.

Tools and Methods of Instruction

The students were usually given time off from their daily responsibilities to attend the course on a full-time basis. Availability of a computer laboratory is absolutely vital for a better understanding of the language and hands-on training. Lectures that exceed 50 or 60 minutes in length are very likely to become tiring. This may cause individuals to lose interest in the subject. The daily schedule should combine a mixture of classroom instruction and laboratory practice using an alternating format. There should also be appropriate breaks between each session. Early in the course only one lecture hour was included in the afternoon. Later, the students spent all of the afternoon hours working on the lab assignments. On the average, the students spent 40% of their time in the classroom and 60% in the computer lab.

All lectures were video taped and were made available to students at the end of each day.

Course Outline

Students had access to several reference books throughout the course. A copy of all handouts in a textbook format was made available to each student. The outline chosen to cover Ada was as follows:

1. Introduction
 - Brief history
 - Ada and Software Engineering
 - Structure of Ada programs, simple types
2. Subprograms
3. Control Structures
 - If stmts., loops, exceptions
4. Scalar Data Types
5. Packages
6. Composite Data Types, Files
7. Access Types, Private Types
8. Generics
9. Tasking

All simple and composite types were covered in detail. Packages, private types, and control

structures were given extra attention. The remaining topics were covered on an introductory level by presenting the major features and structures from each group. At the end of the course the students were given a list of the topics which were not covered. The students gained enough knowledge and training from the course that they could easily learn other language features individually. Some of the topics that were not covered include: tasks with family of entries, pragmas, and low-level I/O.

Assignments

There were two types of exercises: review problems and programming assignments. The review problems were a collection of simple questions that were given to the class every day during the last lecture hour. The purpose of these assignments was to recall the topics covered during the day and to familiarize the student with the syntax and semantics of the language. It also helped non-programmer students become acquainted with Ada code. The problem solutions were made available during the same hour.

Each programming assignment was composed of two or three problems having different difficulty levels. Each student was asked to select a problem and implement it. This method proved beneficial since students did not share common programming skills and the course did not become too demanding for those who were not going to code in Ada. Students felt comfortable with the language as they were not competing against co-workers.

The following is a brief summary of the programming problems that were made available.

Assignment-1

- A. Write a program to compute a person's age.
- B. Implement and test the conversion functions Inches_to_Centimeters, Fahren_to_Celsius, Quarts_to_Liters.

Assignment-2

- A. Using the given algorithm, write the function Square_Root and print the table of square roots for numbers from 1.0 to 10.0 in steps of 0.5.
- B. Write a program to calculate the weekly wages of an employee.

Assignment-3

- A. Using the square root function from assignment-2, write a program to print the windchill factor table.
- B. Convert the conversion functions of assignment-1 into a package.
- C. Convert the payroll program of assignment-2 into a package.

Assignment-4

- A. Implement the package Class_Roster_Pkg to provide resources needed to grade a class of students.
- B. Implement the package Stack_Pkg and provide the common stack operations.

Assignment-5

- A. Using the stack package from assignment-4 write a program to determine if a given piece of text contains balanced parentheses.
- B. Using the stack package and the given algorithm, write a program to convert an infix expression into the postfix notation.

Assignment-6

- A. Write a generic procedure called Array_Search to search an array for a given item.
- B. Modify the Stack_Pkg to use a linked list structure.
- C. Convert the Stack_Pkg into a generic package.

Assignment-7

- A. Write independent tasks to find the max and min of a list of items stored in an array.
- B. Write a program to implement the following tasks:

Input_Task : reads in a list of words from a data file and stores them in a buffer by calling the Buffer_Task.

Buffer_Task : a server task with entries for deposit and removal of data items.

Output_Task : removes data items from the buffer task and stores them in a data file.

Student Evaluations

The attendees of two courses were surveyed to evaluate the overall quality of the instruction. The following summary is the evaluation of a group composed of managers (5%), software engineers (17%), programmers (39%), and other personnel (39%).

According to the survey, 15% of the students found that the Ada history could be eliminated from the course. About 16% were interested in including Pragmas in the course. More than half of the students indicated that tasking and generics should be covered in more depth. More than 65% of the attendees believed it was a good idea to use

the standard I/O instantiations after studying generics (the instructor used pre-instantiations of Integer IO and Float IO for input and output). Half of the students believed the assignments were fair and 28% found them simple. The course length was just right according to 72% and finally, more than 94% of all students believed that the lecture periods were sufficient.

Summary

It is the instructor's experience and the students' overall response that Ada can be taught effectively to industry professionals in short, intensive courses. In order to cover most of the language features, the training period should not be less than 10 days. Other factors that are found to be beneficial in conducting such courses are as follows:

1. The students should be given time off to attend the course; otherwise, job responsibilities can create a lack of time and interest in Ada which defeats the training purposes.
2. A computer facility must be accessible to provide hands-on practice. The instructor should be available in the lab to assist the students with their problems and to provide personal guidance on design issues.
3. Lectures should not exceed 50 minutes and should be followed by lab practices. Total lecture hours during the day should not exceed three. Maximum time for the afternoon lecture should be one hour. To reduce the intensity of the course, short breaks between sessions are highly recommended.
4. If possible, lectures might be video taped to provide the students additional reinforcement.
5. Students do not usually share common skills; therefore, each assignment should provide problems with varying levels of difficulty to meet the needs of every individual.
6. At the end of the course the students should be recognized by awarding certificates of attendance.

Biography

Ferideen Mainian is an assistant professor in the Department of Mathematical Sciences at Cameron University. He received his M.S. in Computer Science from the University of Oklahoma in 1981. Prior to teaching, he worked in industry and was involved in developing educational software. He has designed and taught several Ada language courses to computer science students and to industry personnel since 1987.

Mathematical Sciences Department
Cameron University
Lawton, Oklahoma 73505

Integrating Ada Training with Software Development

Pauline Fortin and Freeman L. Moore
Computer Systems Training
Texas Instruments Incorporated
Dallas Texas 75265

ABSTRACT

Texas Instruments has been providing training to its engineering staff since 1977, with a curriculum that has evolved with time and experience. The introduction of Ada training in 1983 and more recent emphasis on requirements analysis, has provided the impetus to integrate the curriculum of available courses into a coherent software engineering curriculum, covering the full DoD-STD-2167A life cycle. The Ada programming language affects several courses, which we have tied together using common approaches, and common exercises. We see this as vital towards our efforts of maximizing the training resources available to software engineers.

INTRODUCTION

The Computer Systems Training branch within Texas Instruments has been providing needed training for software engineers in the Defense Systems and Electronics Group (DSEG) since 1977. Ada training was first introduced in 1983 with a five day introductory course with hands-on experience. This course has been the staple of the Ada training curriculum and has had an impact on the development of other courses in the software engineering curriculum offered within Texas Instruments.

The introductory course has been supplemented with an advanced topics course, manager's overview, and technical proposal issues workshop. In the interest of maximizing the benefits of training, a comprehensive review of the computer systems training curriculum was undertaken to determine where courses overlapped each other and to determine how courses could better relate to one another.

With a history of separate courses which either present portions of the Ada programming language, or provide overviews, Texas Instruments has chosen to look at the entire set of courses in an attempt to integrate the curriculum available to our software engineers. The analogy to a salad bar has been made about the courses that were available in the past -- that is, the prospective attendee can pick-and-choose what is wanted, without any guarantees of getting a complete and balanced selection. This has been partially resolved by the introduction of training plans which

provide for organized paths for satisfying training needs. We have learned about the needs of the engineering staff during the development of courses, and we are expanding the training opportunities to cover the complete life-cycle. [1]

CURRICULUM

Most of the software development in DSEG supported embedded microprocessor systems running a real-time environment. As a general rule, the MIL-STD-1750A processor is used although other microprocessors are being introduced. The MIL-STD-1750A is a 16-bit instruction set architecture developed by the Air Force. Within DSEG, there are many diverse projects working under an everchanging set of requirements, developed according to military standard specifications and documentation requirements. These projects may involve electro-optics, avionics, missile systems, or software development applications. The primary thrust is real-time systems, with an increasing emphasis on making Ada applications work within embedded systems.

Figure 1 represents the basic training model for DSEG software engineers. Some of the courses support software development skills whereas others involve software engineering training using CASE toolsets.

The Software Engineering Workshop is a three-day course to assist software, electrical, and mechanical engineers who are developing software. The course introduces DSEG software practices, standards, and DoD-STD-2167A [2] life cycle requirements. Participants work in teams to analyze actual project documentation, write portions of a requirement specification, and participate in a Software Specification Review.

A Software Quality Assurance (SQA) Orientation Class is available for software quality assurance engineers after attending the Software Engineering Workshop. This is a three-day course which introduces the attendee to SQA practices in the product life cycle. Participants work in teams to define evaluation criteria, audit test results, trace test requirements to requirements specifications, and evaluate approved DSEG project documentation.

Software Engineering Workshop

SOA
SCM

Introduction to Real-Time systems

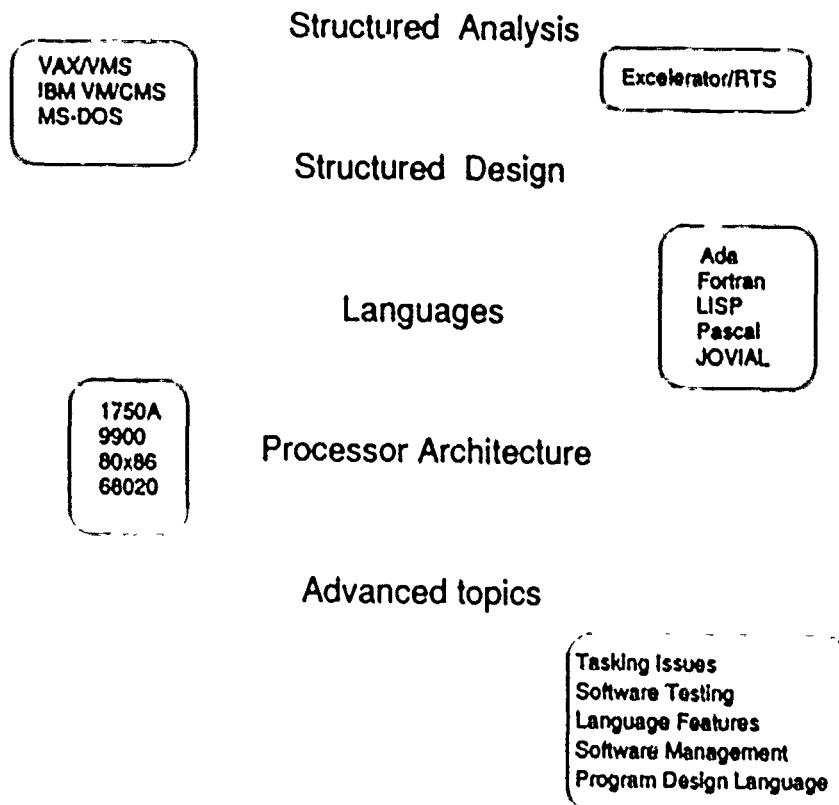


Figure 1

Real-Time principles are fundamental to software development performed within DSEG. Computer Systems Training provides an "Introduction to Real-Time Systems" course to supply the necessary background to those engineers who need it. Other courses deal with the architecture issues of a MIL-STD-1750A processor, as well as how to microprogram array processors that are used by some projects.

"In the near term, says Barry Boehm... most organizations could double software productivity using nothing more than the best software practices and technologies already available." [3] Our DSEG CASE Steering Committee recognizes this importance on standardizing an approach to software development. Our approach is based upon work done by Paul Ward and documented in his books [4]. We now have in place a real-time structured analysis course,

followed by a structured design course. The design course extends the analysis by developing the application model from the analysis course into a top level software design model.

CASE toolsets are gaining acceptance and credibility in the software design community. Software engineers are now using software that costs more than the hardware it runs on. CASE toolsets help automate some of the work involved in documenting the products of structured analysis and design. The toolsets can even provide some help in code generation and producing military standard documentation. The capabilities of Excelsior/RTS are taught in a hands-on course.

The Ada component of the curriculum is perhaps the most stable, yet the most dynamic portion. Grady Booch's book

[5] provides the basis for our introductory course, which has been available for five years. The course has changed support environments, from NYU Ada/ED to Data General, to Dec-Ada, and now, workstations. The courses have stressed proper software development (not simply syntax). With the introduction of the Tartan Ada/1750A compiler, we are in the process of developing additional training aids to satisfy demand for processor specific training.

The Software Engineering Project Management course helps software managers and lead engineers to plan, budget, and control software projects. The course uses case studies and handouts that deal with typical software life-cycle deliverables in conformance with DSEG software methodology and the DoD acquisition cycle. We revised this course in early 1988 to accommodate additional data about managing Ada projects.

CURRICULUM INTEGRATION

To remove the "salad-bar" image of the software engineering curriculum, it has been necessary to review the content of the various courses. By understanding why a course is needed, we are better able to define how the courses can interface with one another. For example, several of the later courses assume knowledge of DoD-STD-2167A for documentation purposes. Beyond having prerequisites for courses, we have chosen to integrate the content of certain courses.

For example, the SW Engineering Workshop introduces the DoD life cycle and documentation requirements by means of a case study. Real-Time structured analysis topics are introduced to lead into the Real-Time Structured Analysis course. The case study used in the Real-Time Structured Analysis, Structured Design, and Excelsior/RTS courses is the same, the low level control of a plotter. The implementation phase of the plotter was recently introduced into the introductory Ada course, adding yet another common thread between the courses.

The Structured Design course introduces people to the concept and notation of program design languages (PDL), which is continued into practice in the Introductory Ada course. With the availability of better workstation compilers, we have chosen to teach four out of the five days using a workstation environment. The last day of the course will deal with interfacing with a VAX environment and its set of tools for Ada development. This represents a recent change from when the course was entirely VAX oriented. The workstation environment we have selected, IntegrAda [6], provides an interface for using PDL notations which is consistent with VAX based tools.

As noted above, the use of one or more case studies has allowed use to reinforce the connections between the various

classes, and provides a consistent application from which the student can learn. Starting from the analysis phase through the implementation phase, students are exposed to all facets of a problem with which they have become quite familiar. Some of the common threads are identified in the appendix.

INTEGRATING ADA INTO THE CURRICULUM

As previously mentioned, Ada has been a part of the Computer Systems Training curriculum for several years. With a review of the entire software engineering curriculum needed, a review of the Ada curriculum is also in order. This was facilitated in part, by the introduction of two additional instructors, who offered their fresh perspectives on the content and role of the Ada courses.

In 1988, we added a real-time structured design course to our curriculum. Introducing this course has required us to review the approach that Booch takes to object-oriented design in his book. It was necessary for us to refine minor portions of both courses to make the content flow consistently from one course to the other. Our approach is to continue the use of the transformation schema and structure charts from the Analysis/design course, refining the information content into "booch-grams" used in the Ada course. We have resisted introducing new graphical notations due to the nature of the CASE tools we have available [7].

The concept of a program design language (PDL) is used in the structured design course as a means of writing process specifications. With the use of IntegrAda in the Introductory Ada course, we are able to continue the use of PDL annotations easily as part of developing solutions to exercise problems. The Advanced Ada course continues this emphasis on design notation by utilizing the AdaDL toolset to produce a software design document. AdaDL is an Ada Program Design Language supported by VAX hosted tools [8].

The importance of DSEG Programming Standards is identified in the Software Engineering Workshop, Software Quality Assurance workshop, and reinforced in the Ada classes with attention drawn to the content of the standards. Discussion is also raised about the motivation of some of the standards, such as restricting the *with* and *abort* statements.

We find it necessary to integrate the role of the life cycle with software development as practiced in the Ada classes. This is evident in the structure of the exercises developed, starting with analysis, top-level design, detailed design, and then implementation. One of the programming exercises is a modification of existing software, allowing them the opportunity to work at the maintenance phase as well.

Our introductory Ada course covers the entire Ada programming language in five class days. The programming labs include time where students have an opportunity to exercise most of the language features. We have taken the approach of letting the Advanced Ada course deal with efficiency issues. We find this approach useful since students will have learned the complete language in the Introductory Ada course, and now are refining their design and implementation skills in the Advanced Ada course. This is carried a step further in our Ada/1750A training by addressing language/processor specific details in depth in this material.

One of the goals of our Ada training is to continue the involvement of the instructors beyond the classroom environment into the project world. This benefits both instructor and project. The instructor is kept up to date with how material is being applied on the various projects. The project benefits by having an outside Ada expert available for candid review of software and documentation.

ASSESSMENT OF TRAINING EFFECTIVENESS

Most courses have exercise components to reinforce concepts discussed in lecture. To encourage the completion of exercises, completion certificates are provided to those that successfully finish the work. The class schedule provides time to finish work during class, although a few people may require extra time in the Ada courses to complete the work. This has been accomplished in the past by making the necessary computer accounts available for one additional week after the class.

The effectiveness of our training activities is measured by several means. First is the informal observation by the instructor of the class. The questions raised by the people often provide the necessary clues as to the quality of the training, and the on-the-spot customization often needed when dealing with classes with diverse backgrounds. Projects have requested special offerings of the Ada training courses which has required additional preparation by the instructor to have the training reflect the needs of the project environment.

Toward the end of each class delivery, a standard Participant Evaluation Form is passed out to the attendees to complete in a multiple-choice fashion. Written comments are encouraged, but often are not collected in an organized fashion. The Participant Evaluation Form is machine scored, with results being merged into a department database. By looking at the results over a period of time, trends can be detected and acted upon if necessary.

We have started to introduce the concept of pre and post-testing in some courses. We see this as the opportunity to document that a course is achieving a training goal in a measurable sense. This work was recently begun and will continue to expand.

LEARNING FROM OUR MISTAKES

Our curriculum has changed over time. For instance, the courses dealing with Pascal that were taught just three years ago, are no longer taught due to our emphasis on a single language, Ada. It took us too long to develop a coherent picture of software development, and integrate it with the work environments, starting with PC-based work through VAX-based development.

When the Introductory Ada course was first introduced, it was offered as a five day course, presented in one week. We have since modified this delivery schedule to have the same five days spread out over a two week period, meeting every other day. We have found the extra day between class days reduces the impression that a great deal of information was being presented. The course content is the same, just delivered over two weeks time instead of one.

We have changed our examples and exercises to reflect project needs. Our students are here to learn how to apply the language back on the job. Having relevant examples is one way of having the course reflect the job requirements. Another way is to make sure the software training environment is the same as the work environment. We find projects are switching to workstations for software development. This has been reflected in our Introductory Ada course switching from the VAX environment to a workstation for training purposes. We have kept in an exercise dealing with the VAX environment to satisfy those students who have the requirement of doing software development on a VAX.

A group level decision about the role of training has made a substantial impact on the direction of our work. Without this direction, training was primarily used to meet an immediate need. With training plans in place, we are seeing a more organized approach to raising the skill level of a larger group of software engineers, instead of isolated groups as before.

FUTURE WORK

To complete the training from a life-cycle perspective, we need to fill a gap in integration/testing. We shall be attacking this problem in 1989. We are continuing our development of training for Ada as implementations become available on various processors, including the TMS320C30 signal processor.

Our goal has been to integrate Ada training with our software development curriculum. We believe that we have essentially met this challenge. We found that several courses were affected in the process. Integrating the courses meant more than adding prerequisites. The greatest challenge is not in providing the best course on Ada training, but rather making sure that Ada is properly used as the vehicle towards building better engineered systems.

REFERENCES

- [1] Connolly, Daniel, "An Ada Training Life Cycle Curriculum", 6th National Conference on Ada Technology, March 1988.
- [2] DoD-STD-2167A, *Defense System Software Development*, Department of Defense Military Standard, 29 February 1988.
- [3] Newport, John, "A Growing Gap in Software", *Fortune*, April 28, 1986.
- [4] Ward, Paul & Mellor, Stephen, *Structured Development for Real-Time Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- [5] Booch, Grady, *Software Engineering with Ada*, Menlo Park, CA: Benjamin/Cummings, 1987.
- [6] IntegrAda, product of AcTech, Inc, Solana Beach, California.
- [7] Alaiso, Bruno, "Transformation of Data Flow Analysis Models to Object Oriented Design", OOPSLA '88 Proceedings, September 1988.
- [8] AdaDL: Ada-Based Documentation and Design Language, product of Software Systems Design, Claremont, California.

APPENDIX: Common Threads Between Courses

Software Engineering Workshop:
Introduces DoD-STD-2167A, project background, need for standardized software development.

Real-Time Structured Analysis:
Relates methodology to 2167A environment, defines steps to follow, introduces case study, and exercises.

Real-Time Structured Design:
Continues structured analysis methodology using same case

study and exercise. Introduces PDL notation to describe processing components.

Excelsior/RTS Fundamentals:
Implements notation of structured analysis/design on Excelsior/RTS toolset using a common example from the SA/SD course, a Plotter.

Fundamentals of Ada Design and Programming:
Object-orientation is followed from the structured design course, with implementation of the Plotter which was used in the analysis, design, and Excelsior/RTS courses. PDL notation is used, following on the notation introduced in the design course.

Advanced Topics in Ada:
Relates topics back to system analysis and design concerns, reinforcing object orientation. Provides greater detail on the use and benefits of PDL as a means to describe processes.

ABOUT THE AUTHORS

Pauline Fortin is a member of the Computer Systems Training Branch of the Human Resources Development Department at Texas Instruments. She has a Master of Science degree in Computer Science from North Texas State University. She has been involved in the development and delivery of courses within the software engineering curriculum. She is the lead engineer for the Software Quality Assurance, Software Engineering Workshop, and Introductory Ada courses.

Mailing Address:
P.O. 650311, M/S: 3928
Dallas, Texas 75265
e-mail: fortin@flopn2.ti.com

Freeman Moore is also a member of the Computer Systems Training Branch. He has masters degrees in Mathematics and Computer Science from Central Michigan University. He is completing his doctoral courses in Computer Science at the University of North Texas. He has been involved in the Ada training curriculum for the last four years, with recent attention being given to integrating real-time structured analysis/software design training capability into the software engineering curriculum. He is the lead engineer for Advanced Ada and the Real-Time Structured Analysis / Software Design courses. Both authors are working on the development of processor specific training for Ada and maintain heavy involvement with the project community within Texas Instruments.

Mailing Address:
P.O. 869305, M/S: 8435
Plano, Texas 75086
e-mail: fmoore@skvax1.ti.com

EVALUATION OF TEACHING SOFTWARE ENGINEERING REQUIREMENTS ANALYSIS (SERA)

Jag Sodhi

TELOS Federal Systems
Lawton, Oklahoma

ABSTRACT

Understanding and analyzing a customer's requirements for a system's software engineering is the primary focus of this paper. This paper evaluates the teaching of the Software Engineering Requirements Analysis (SERA) Course for understanding and analyzing the customer's requirements of real-time, embedded systems. The systems software engineering are being developed in Ada for US Army Communications-Electronics Command (CECOM), Fire Support Systems, and Life Cycle Software Engineering Centers (LCSEC).

INTRODUCTION

A system needs analysis first including proto-type modeling before starting the system's design - Preliminary design and Detailed design. The SERA course uses structured techniques to graphically analyze customer requirements. The techniques separate the requirements into manageable logical independent functions. The relationship of functions and objects are then established for use later on by the Object Oriented Design Method (OODM) and Ada advanced features. The course covers software engineering goals and principles. The course discusses various phases of the software engineering life cycle development phases in accordance with DOD-STD-2167A as illustrated in Figure 1.

PARADIGMS OF SERA

SERA has gone through many revisions to include appropriate work-related examples, exercises, and case studies. A generic real-time example as shown in Figure 2 is discussed in detail along with many exercises and case studies. Upon completion of the course, the students understand the various phases of software engineering, the importance of system modeling, (modeling assists in importing pre-tested reusable Ada packages from other related systems), structured technique to analyze requirements, and possess at least one view of the "blue print".

The students gain practical experience in performing structured walk-throughs. Exercises are included to give the student practice in analyzing solutions with the help of tools. Tests and quizzes are used to measure the students' progress and achievement.

This paper also discusses how SERA can be automated with the help of Computer Aided Software Engineering (CASE) tools. This is helpful in creating quality and reducing costs of the system software engineering products. Thus the systems produced are more reliable and well documented.

CHARACTERISTICS

SERA course was designed on the basis of the structured approach to understand customer requirements and translate correctly these requirements to develop software efficiently and cost effectively to the customer's satisfaction. Software engineering is to be maintained for a life cycle with frequent changes implemented from the users.

The first SERA course was experimented with by selecting a few professionals to walk-through the material. The lessons learned were included in future revisions.

The course was revised to suite the need of TELOS. This was used for the in-house education and training of professionals rather than only an educational course. The expectation was that students learn from hands-on training which benefits their work. This concept is different than the commercial courses available in the market where the students are provided only exposure to the subject in five days.

The course was originally prepared to be offered for five full days duration. Many professionals had time constraints to finish their assigned tasks. Then classes were offered ten half days with a lecture in the morning and workshop in the afternoon so the student can take the assignment back to the office to complete. I experimented with all of these alternatives as suggested by the students. All of these suggestions had some good points. Finally, I accepted the majority consensus to teach the course for three full days with the commitment that the students will accept and complete the workshop as homework. This technique is working very well. The company is saving time and money. Each student willingly accepts homework and spends their extra time to complete it on time. Some of SERA teaching characteristics are:

- Functionality
- Follows set standards
- Structured approach
- Accuracy
- Consistency
- Modifiability
- Produce quality results
- Easy to understand and follow

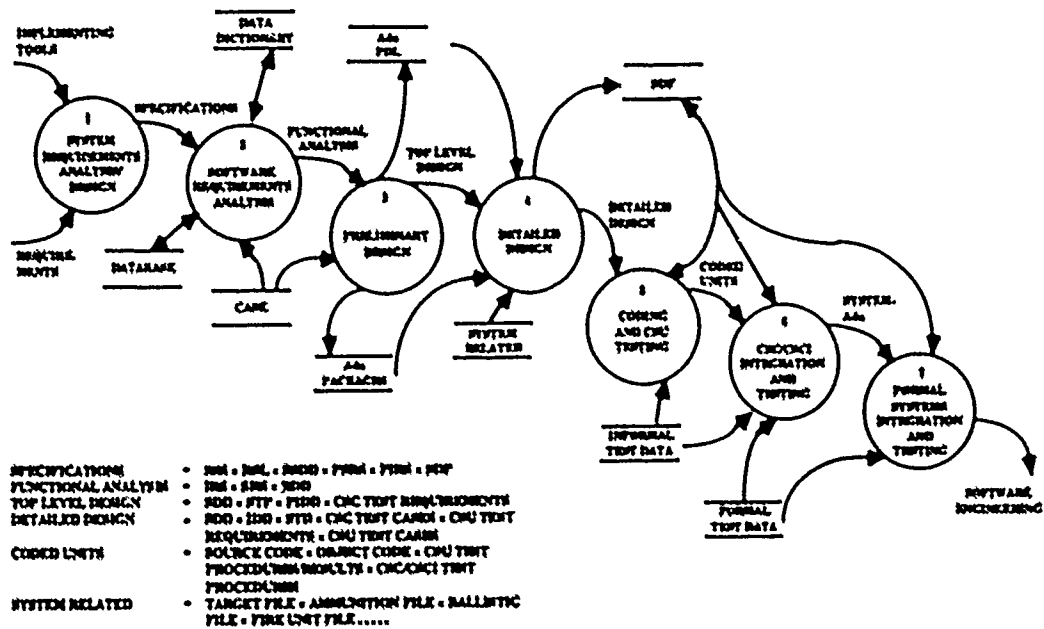


Figure 1: Overview - Software Development Phases

CORTEXT: "Managing Ada Projects Using Software Engineering" J. J. Smith, Prentice Hall, Inc., NJ

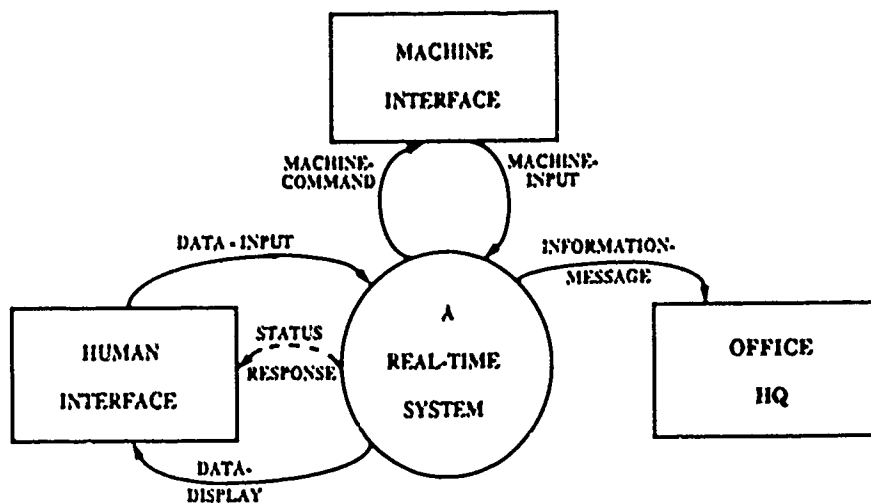


Figure 2: A Real-Time System Context Diagram

METRICS AND INDICATORS

Metrics are used as a means for quantitatively assessing the characteristics of teaching SERA. The metrics are indicators that objectively assess the quality of teaching and determine how much knowledge the students have absorbed. These indicators aid in obtaining valuable feedback from the students. The student evaluation is carefully designed to receive the correct assessment of the course as shown in Figure 3. Some of the metrics and indicators introduced are:

- Tests
- Quizzes
- Mini exercises
- Students make their own work related case study for the final examination
- Walk-throughs
- Open discussions

STUDENT EVALUATION SHEET

COURSE: TELOS Software Engineering
Requirements Analysis (SERA)
(By: Jag Sodhi)

DATE:

NAME: DEPT/SECTION: _____

You can make a personal contribution to this effort
by answering the following questions:

1. What did you like most about the course?
2. What did you like least about the course?
3. What would you like to see deleted from this course?
4. What would you like to see added to the course?
5. Are you willing to adopt these tools in your work?
6. If no to #5, please give reasoning why not?
7. Any additional comments?

Figure 3: Student Evaluation Sheet

PARAMETERS

The parameters are characterized among the professionals for the knowledge gained from the SERA course. They have many years of hard experience of trying to mutually understand the customer's requirements for developing software. After course completion they appreciate SERA's usefulness in making their jobs' performance easier. Teaching SERA is not only to share the knowledge but to infuse into the

professionals this knowledge so that they can utilize these tools on the job. The parameters summarized are:

- Professional satisfaction
- Management Return of Value (ROV)
- High morale among the professionals
- Positive feedback from the students
- Increase of quality products on the job
- Build team spirit in the company
- Improvement of the individual's job
- Effective use of time and resources
- Increase cooperation

EVALUATION

SERA has been successfully taught to over one hundred fifty professionals at TELOS for the last two years. Each one of these professionals has evaluated the course very high. The evaluations have many positive suggestions which have assisted in course revisions for making it more presentable and acceptable by professionals.

The concept of student participation was stressed throughout the course. The class was limited to six students for effectiveness. This scheme leads to a more personal touch between the students and the teacher. The class was further divided into two teams to introduce the team concept. Each member first tried the case study himself before having the walk-through with other members of the team. Each team selected their leader for a case study to present the solution on the board. The other members of the team walked-through the solution. The team members assisted and supported their presenter. This not only created healthy competition but also aided the students to learn faster in a shorter time.

All available tools of educating and training are used in the classroom. These tools are the vu-graph (overhead projector), board, and hands-on computer training. I have always believed in teaching a little, then put that teaching into practice by showing an example, and finally providing an exercise to cement the training.

At the beginning of class, I introduce myself and request each student to provide a brief job introduction. This also helps me adjust the teaching for that set of professionals. I explain the goals and objectives of the SERA course, a brief history of how this in-house education and training initiated and what I am expecting from my students. The criteria of their performance in the class will be evaluated as follows:

- Positive attitude
- Work as a team member
- Constructive comments
- Raise constructive positive questions
- Active participation
- Help other members of the team

- Communicate for productivity
- Finish homework
- Promptness/punctuality in attending the class and coming back from lunch and breaks

The students evaluate the SERA course; how the instructor presented the course. Is the instructor knowledgeable enough to answer all their questions? Is the course useful to meet their job needs? Are the examples, exercises, and case studies related to the job? Are they willing to support the program? Do they want any change in the material or the program? The evaluation of the students are statistically compiled and analyzed as follows:

- Teacher
 - Open end presentation
 - Gets everyone to participate and contribute
 - Knows the subject
 - Excellent presentation
 - Great enthusiasm to motivate students to learn the subject
 - Adjust with the class
 - Well prepared
 - Right logical approach
 - Positive attitude
 - Quality instructions
- Material
 - Excellent material
 - Logical format
 - Structured
 - Well organized
 - Quality
 - Practical case studies
 - Clear explanations
- General Comments
 - Excellent course
 - Good atmosphere
 - Doing exercises instead of only lecture
 - All members of the class learned SERA
 - Relaxed atmosphere
 - Smooth continuity
 - Teacher presentation and material tie together
 - Group discussion
 - Student interaction
 - Motivated to learn by doing exercises and case studies
 - Real education and learning

LESSONS LEARNED

I found that there are four basic different types of students:

- Innovative
- Analytical
- Inquisitive
- Practical

The innovative students are concerned with gaining personal knowledge through discussions and interaction, and asking questions to learn "WHY". The analytical students seek the facts about an issue by analyzing ideas or knowing and learning by asking "WHAT". The inquisitive students independently carry on the case studies themselves through to completion. They learn by trial and error and self-discovery. They gain knowledge by asking "IF I DO THIS, WHAT WILL BE THE FINAL RESULT". The practical students need to know how things work and learn through hands-on experiences. They believe in the practical application of ideas, and learn by asking "HOW DOES IT WORK". It has been a challenge for me to satisfy all these types of professionals.

I have found that students like the approach of developing their own work related case study. They select a requirement and try to write an understandable case study. Often one hears in the computer industry that the customer's requirements are not understandable specially when the project is falling or schedules are slipping. I encourage students to create their own meaningful, understandable requirement. This approach assists students to understand their own requirements and produce solutions as they progress and learn in the class. On the final day of class the students are given a chance to present and walk-through the solution with other members of the class.

I experienced that the selection of students to form a class is important. These students are selected from different sections so that they can learn from each other's experiences. Then the class is divided into two teams of three. The teams are divided in such a way that not all members of the team belong to the same section/department. This scheme provides a variety of student experiences to be shared with each other. This also creates a good atmosphere and is a very effective learning result as evidenced by student evaluations.

I learned that the students prefer to have an individual work station. This plan assists in using hands-on training on automated tool kits in the workshop more effectively. The work station should have a good Ada compiler, Debugger, Configuration Management, Editor, and Backup facilities. There are many Computer Aided Software Engineering (CASE) tools available in the market which are suitable to one's requirements. I use Yourdon automated tool kit for my classes.

CONCLUSION

The teaching of SERA follows easy methods to help computer professionals to understand customer requirements. The requirements are translated into graphic models to convince the customer easily that their requirements are understood and the software will be developed in accordance with DOD-STD-2167A in Ada. The teaching lays out, in clear detail, exactly how the customer requirements should be mutually understood. By understanding the requirements professionals take a smooth journey through the software development phases. This approach saves time and money. And most important of all, as the journey progresses, professionals and the customer feel more and more comfortable and gain confidence in implementing the software engineering.

I acknowledge that the evaluation of all my students, who have attended SERA course, has assisted me in building a course worthy of presentation. I treat every class as a project to be completed in time and successfully.

REFERENCES

1. Sodhi, Jag, Computer Systems Techniques, Petrocelli Books, Princeton, New Jersey, (In Printing).
2. Sodhi, Jag, Efficient Techniques for Analysis, Design, and Programming, Scientific and Business Systems, Course Notes 1982.
3. Sodhi, Jag, A Methodology for Software Engineering Development Life Cycle, unpublished, 1986.
4. Sodhi, Jag, "A Handbook for Structured Walkthroughs", TELOS, 1986.
5. Sodhi, Jag, Michael D. Sapenter, "Management Aspects of Real-Time Systems in Ada", SUNBELT SIGAda Conference, November 1988.
6. Sodhi, Jag, Software Engineering Requirements Analysis (SERA), TELOS, 1987.
7. Sodhi, Jag, Software Engineering Design (SED), TELOS, 1988.
8. Sodhi, Jag, "Overview of Ada Features for Real-Time Systems", Defense Science, December, 1988.
9. Sodhi, Jag, Managing Ada Projects Using Software Engineering, Petrocelli Books, Princeton, New Jersey, (In Printing).
10. Sodhi, Jag, George, K. M., "Objects with Multiple Representations in Ada", Seventh National Conference on Ada Technology, 1989.
11. U.S. Dep. Defense, Military Standard, Defense System Software Development, DOD-STD-2167A, Washington, D.C., 29 February 1988.



JAG SODHI has a Master Degree in Mathematics, a Degree in Telecommunication Engineering, and is a Graduate of IBM in Data Processing. He has many years of Data Processing experience in business, financial and scientific applications in various EDP machines and languages. He has conducted numerous professional classes and seminars on these subjects. His publishing credits include numerous training courses on Ada and software engineering. Jag is a senior system engineer and in charge of education and training at TELOS Federal Systems, Region 1.

Ada Abstract Data Types—the Foundation of an Interactive Ada Command Environment

John A. Thalhamer, William P. Loftus, Charles L. Oci, Ralph A. Foy

Unisys Defense Systems
Paoli Research Center

Abstract

A set of Ada abstract data types (ADTs) is the underlying substrate that defines a common, Ada-oriented interface to diverse host environments. The Ada ADTs, in conjunction with the use of Ada as a command language, serve as a unifying concept in the description of a portable Ada command environment. The benefits provided by ADTs and Ada in a software engineering environment are extended into the command language arena. The Ada Command Environment (ACE)¹ combines the power of Ada as a command language with the description of the host environment through ADTs. ACE presents to the user a consistent Ada-oriented, development environment that supports a uniform interface across a heterogeneous set of development architectures.

1 Introduction

There are many Ada development environments available today that function on a variety of platforms. These platforms consist of both hardware and the necessary support software. Each Ada development environment provides an interface through which its Ada tools are invoked and controlled. Each hardware and software platform provides at least one command interpreter through which the many facilities provided in the host environment are accessed and acquired. Associated with a command interpreter is the command language which defines the names, syntax, and interface semantics of the available commands. This combination of Ada environments and host operating environments presents a diverse, formidable set of paradigms. The developer must be familiar with the conventions of both the host environment and the particular Ada environment. The differences between these environments makes it difficult for an Ada developer to easily move between different host environments.

Ada compilation environment builders have taken different approaches in designing the interface to their Ada tool

¹The work described herein was performed under Office of Naval Research contract number N00014-87-C-0743.

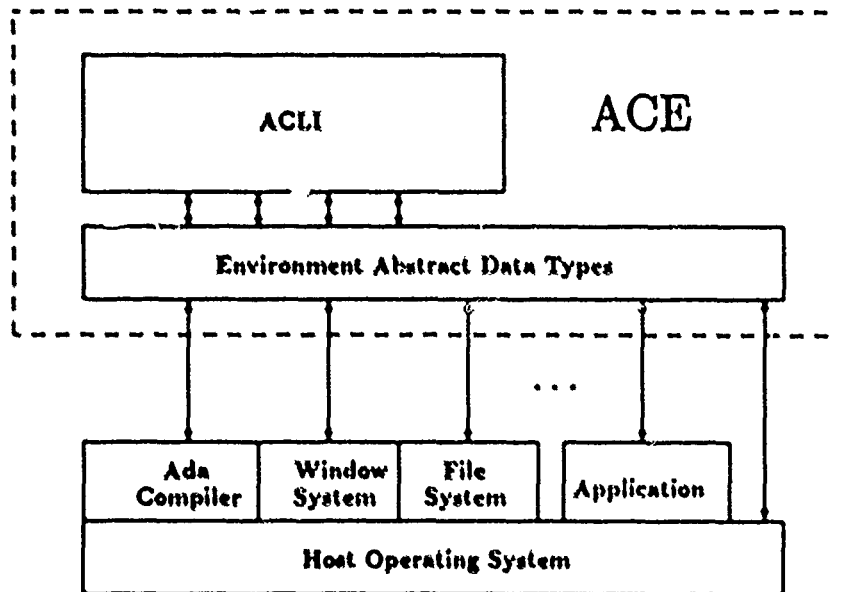
suites. Some environments have adopted the style and syntax of the underlying command language.[6] This approach blends their tool suite with one of the command languages supported on the host machine. The interface to this compilation environment may then change when the identical Ada environment is available on a different hardware/software platform. Other environments have adopted the style and syntax of the Ada programming language.[5] This approach incorporates some of the features of Ada (e.g. procedure call) into the command language. With this approach, a more uniform Ada compilation environment may be available on a diverse set of platforms.

However, interfacing with the Ada compilation environment is only one of the typical tasks performed by an Ada developer. Some of the other tasks performed include the management of file objects (e.g. create a file, list a directory, type the file), the interaction with a diverse set of applications (e.g. electronic mail, configuration management, editor) and creation of scripts to easily perform repeated sequences of tasks. For these tasks, the developer must describe the action to be performed via the command language provided by the host system. The developer must leave the paradigms of Ada and the Ada development environment and adopt the paradigms supported by the command language.

Rather than require the developer to learn and operate under two different paradigms, Ada and that of the particular command language, the Ada Command Environment supports a single paradigm—Ada. Ada is both the programming language and command language with ACE. The paradigm of Ada that is familiar in the programming development environment is also the mechanism used to interact with the underlying system in the other various tasks performed by the developer.

2 ACE Overview

The Ada Command Environment[3] is an interactive, object-oriented command language environment for Ada software development. ACE supports Ada as the programming language and the command language. The command environment of ACE is defined through a set of Ada ab-



Ada Command Environment

abstract data types which encapsulate the host environment into an Ada framework. ACE provides a portable common command language user interface that currently runs on Sun workstations and the Unisys PW2/300—an Intel 80286 based personal computer running MS-DOS. With Ada as the command language, the environment may be easily extended and tailored through the definition of additional ADTs via Ada statements.

ACE is composed of two logically separate parts—an Ada interpreter and a set of Ada packages that define the environment through abstract data types. The Ada command interpreter provides an interactive execution of Ada statements and compilation units. The interpreter in ACE serves as the engine for processing the command language and driving the Ada software development process. Ada, as a command language, is used to invoke operations and to manipulate environment objects. Environment objects are expressed as Ada data types, and commands are expressed as operations on the data types. Ada packages are used to group data types and operations into abstract data types.

ACE allows a common set of Ada-oriented objects and operations to be defined and implemented on a diverse set of platforms. ACE's abstract data types encapsulate the objects and operations into appropriate user abstractions. The implementation of ACE's ADTs allows ACE to be easily ported to a heterogeneous set of host environments.

3 Abstract Data Types and ACE

Data abstraction, information hiding, modularity, and locality are some of the the modern software engineering

principles used in the development of software applications.[1] The notion of data abstraction is also a powerful mechanism for the definition of a command environment—an environment that contains a set of objects upon which a group of command operations act.

An *abstract data type* is an abstraction mechanism that encapsulates a set of values together with a set of operations that apply to the values.[2] Within software development, the decomposition of the system may be defined through a set of objects, the operations applicable to the objects, and the operations needed by the objects. ADTs serve as a natural description method for this type of system decomposition. ADTs are also a key component of the object-oriented design and development approach.

The directives issued by a software developer to the underlying host environment may also be naturally defined through the use of ADTs. Each directive or command may be viewed as an operation; the qualifiers or parameters may be viewed as the objects upon which the operation is performed. Logically associated objects and operations may be gathered together into collections which are related to particular components of the underlying host environment. Thus, a parallel can be drawn between abstract data types and the composition of a command language.

Many of the newer procedural languages provide syntactic mechanisms to easily specify and manipulate ADTs. Ada is one such language. The constructs of packages (specification and body), subprograms (functions and procedures), subprogram invocation, type declarations, object declarations, and context clauses are examples of Ada's support for ADTs. The Ada Command Environment makes use of these Ada constructs to define the environment objects and operations through ADTs.

ACE provides an Ada ADT interface to the underlying host environment in the form of Ada package specifications. The package specifications are processed by ACE upon initiation. Thus, a set of predefined types and operations are made available to the user from the beginning of an ACE session. Since these types and operations are defined via the Ada package construct, the methods used to manipulate Ada packages are also used to manipulate the operation of the environment ADTs.

4 Benefits of the ACE Approach

The combination of ADTs and Ada provide many benefits in a command environment. Ada provides a strong language foundation for the construction and use of ADTs, and ADTs provide the mechanism for environment manipulation. The following sections describe the unique features (above and beyond normal command languages) of the ACE approach.

4.1 The Ada Language Standard

Ada, as a modern procedural language, encompasses many of the state-of-the-art software engineering principles. These principles are extended into the command environment through the use of Ada to define the environment with ADTs.

The Ada package construct supports the principles of data abstraction and information hiding through the separation of the package specification from the package body. The separation of the specification and implementation of the abstract data type in Ada and ACE is a key element in the ability to tailor the environment. Different implementations of an environment ADT specification are an obvious mechanism for tailoring the environment to a project's taste. For example, a common configuration management interface may be defined through a single ADT specification, but different implementations may be written based upon the project's particular selection of a configuration management application system.

The ability to layer ADTs within Ada supports the principles of modularity and locality. Environment extensibility may be accomplished through the use of layered ADTs. For example, a new ADT specification may be written that presents an interface that is more familiar or comfortable to the user. The implementation of that ADT simply invokes the standard set of operations. The ADT makes the translation from user orientation to system orientation, rather than forcing the human to mentally perform the translation. Layered ADTs also support the notion of different levels of abstraction. For example, the notion of formatting a textual document, building its table of contents, and

printing the result on a printer may be viewed as either a single operation or a series of lower level operations. Low level ADTs serve as the building blocks for higher level ADTs.

Within the language definition of Ada, Ada is used to extend its own definition. The Ada input-output operations (chapter 14 of the reference manual[7]) are provided in the language by the means of predefined packages. In addition, other predefined library packages are required for each Ada implementation. ACE has implemented the Ada predefined packages, such as `Standard`, `ASCII`, `Calendar`, `System`, and `Text_IO`. This set of packages makes the standard Ada types and operations available in the command environment. Continuity is established between the command environment and the typical Ada development environment.

ACE also views the set of Ada predefined packages defined in the reference manual as a set of guidelines to be followed in the development of environment ADTs. The input-output packages of chapter 14 of the reference manual[7] denote a style of operation definition and manipulation that ACE has expanded to encapsulate the entire environment. The `Create`, `Open`, `Close`, and `Delete` procedures that are applicable to file objects are used within the command environment to define similar control operations upon other types of objects. An example of this is the similar treatment of file objects and window objects. File objects and window objects are each abstract data types in ACE that are created using the `Create` procedure and removed using the `Delete` procedure. The operations that the Ada developer is familiar with in the program development environment are the same operations that are to be invoked within the host environment to accomplish similar tasks.

The guidelines are followed in more detail than simply through subprogram names. Names and modes of parameters, the selection of a procedure versus a function, and the use of the `Form` parameter as a string data type to specify non-default implementation options are all further examples of following the style of Ada as defined in the language standard. These and other instances of conformance within ACE, enforce an Ada-oriented style of ADTs within the ACE environment.

4.2 Command Structure

Consistency and uniformity in the command environment of ACE is achieved through the use of Ada and ADTs. Commands and objects are logically grouped together as ADTs via the Ada package mechanisms. This grouping allows the environment to be structured and ordered. In addition, by nesting packages and subprograms the environment provides controlled access to information. Users explore the environment in an orderly and informative man-

ner. This logical grouping of environment components has many benefits over the flat structure supported by most command languages.

For example, if a specific windowing package is nested inside a basic windowing package, novice users must "use" or reference the basic windowing package before they can access the specific windowing package. This does not guarantee that novice users understand the environment. However, it does guarantee that novice users understand the logical structure of the environment. Of course, expert users who know the structure of the environment are not hindered, since they can simply reference an arbitrarily nested command via the Ada expanded name feature.

Another benefit of this command structure combined with Ada is the ability to define a user interface that is consistent with the paradigms of Ada, as well as uniform in its treatment of objects and operations in the environment. Such an environment would support (at all levels of interaction with the environment) Ada philosophies, providing an excellent vehicle for Ada development. The facilities of overloading and derived subprograms in Ada provide the opportunity to define uniform interfaces to logically related operations and objects. As described above, the ability to define a Create operation for each type of environment object is supported in Ada through overloading. ACE supports overloading to allow the uniform definition of abstract data types across the entire command environment. In addition to being consistent with the Ada standard, the environment is also uniform among the ADTs that are defined within it.

4.3 Command Applicability

One benefit of modern procedural languages is the notion of strong typing. The benefits of strong typing within Ada are also of benefit to Ada as a command language and the definition of ADTs. While ADTs allow the definition of operations for objects, strong typing enforces the proper use of the operations. Many of the problems associated with a novice's use of a command language can be attributed to the application of operations to inappropriate objects (e.g., printing a binary image). In a strongly typed command language, and in particular ACE, if there is no operation "print" defined for binary image objects then the user can not (even accidentally) apply the operation.

Another benefit of strong typing in a command language is in the operation of very large software systems. Many of the benefits of using ADTs in the construction of these software systems are retained in the command language which acts as the "glue" which holds such systems together. Having a strongly typed command language helps guarantee that the systems are correctly constructed from their components. In addition, having a compilable command language allows an interpreted system to become an en-

tirely compiled system merely by compiling the command language, whereas in a traditional command language, the "glue" would have to be rewritten into the system's programming language.

4.4 Command Specialization

Through the use of derived types and derived subprograms, new objects can be described as specializations of existing objects, i.e., described as differences from existing objects. For example, the entire abstract data type for ACE's hierarchical file system is constructed of existing ADTs that are specializations of a general file ADT. The general file ADT provides the basic operations (e.g., Create, Delete, Copy, Rename, etc.) that can be performed on all files. The immediate specializations of the general file ADT are TextFiles, DirectoryFiles, and BinaryFiles. Each of these specializations provides specific new or redefined operations for each type. Any operation defined for the general file ADT that is not redefined in a specialization's operations is inherited by the specialization. Therefore, each specialization of the general file ADT inherits the Create, Delete, etc. operations, which in turn allows every type of file in the file system to be manipulated via the general file operations. Specialization provides a very powerful reuse mechanism within ACE; existing objects can be extended or tailored for particular applications or user aesthetics without having to describe the entire ADT.

In addition, since Ada (and consequently ACE) implicitly derives subprograms for every derived type, much of the work that is normally associated with strong typing in a command language and the construction of a hierarchical command environment is removed from the user. Each derived type implicitly inherits a set of commands that enable its basic manipulation.

4.5 Command Extensibility

An important part of any state-of-the-art environment is the ability of the environment to evolve as technology and methodologies evolve. ACE's approach is to use Ada ADTs to define the command language (creating a command environment). As described before, Ada ADTs have a clean separation of implementation from specification. Therefore, as technology makes small leaps, the new techniques can be incorporated in the ADT implementation while not effecting the specification. In addition, when radical breakthroughs are made in technology, new environment ADTs can be constructed and incorporated into the command environment. Using this approach, we are only limited by the ability of Ada to assimilate new approaches.

5 ADT Interfaces within ACE

Abstract data types within ACE are defined by Ada packages. The package specifications encapsulate the definition of the objects and the operations that are applicable to the objects. Additionally, the package specification provides a mechanism for information hiding particularly, hiding of the operations' implementations. The Ada package body contains the implementation of the object and its respective operations.

ACE supports two mechanisms for the implementation of the ADT bodies: *interpreted* and *built-in*. Both of these mechanisms support a different facet of environment definition, and together they provide the facilities to compose and extend the Ada command environment. Additionally, ACE through its ADTs provides a mechanism to access executable images external to ACE. This provides added power and flexibility to the command environment.

5.1 ADT Body Implementations

As previously stated, Ada is the command language accepted by ACE and interpreted by ACE's command language interpreter. The environment (as defined as Ada packages) is read by the command language interpreter and processed, resulting in the elaboration of Ada packages. This process of interpreting Ada ADT package specifications and bodies is the typical method through which ADTs are declared within ACE.

ACE provides an additional mechanism by which package bodies may be defined. Rather than interpreting an Ada package body, the Ada code may be compiled and linked into the ACE executable. The package specification for the package is still Ada code that is interpreted by ACE. A pragma directive informs ACE that the package body associated with this package specification is already compiled and included within ACE.

This method of package body inclusion provides benefits to the runtime efficiency of ACE. ACE may be tuned such that frequently invoked code is executed at the machine language level (i.e., the compiled level), rather than interpreted.

Another benefit of compiled implementation is that it provides interactive invocation and composition of compiled code within the command environment. An example of this is the X Window System ADT of ACE which provides Ada interfaces to the X Window System (currently implemented in C; see section 6).

5.2 External Images

A vast array of applications and support tools are typically available within the host environment. ACE does not impose a restrictive environment that limits the facilities available to the software developer. Through a host operating system ADT, ACE provides an interface mechanism which makes external executable images on the host system available from within the command environment. Thus, environment ADT specifications are able to define a consistent Ada paradigm for the user that may interface with a diverse set of Ada and non-Ada external images, including the host operating system.

The ability to access external images provides the opportunity to build high level, Ada abstractions from low level, non-Ada applications. Relationships may be formed among stand-alone applications, providing a higher level data abstraction that encompasses the user's desired functionality. The intricacies and/or idiosyncrasies of the individual applications are hidden from the user in the ADT implementation. The implementation also hides the handling of intermediate results being passed between applications. The user simply sees the specification, which is designed to provide a consistent interface within the Ada-oriented environment.

By invoking external images through environment ADTs, the functionality of ACE can be extended into domains which can be tailored to specific environments, projects, or users. For example, a project-oriented configuration management ADT can be defined which provides software configuration control objects and operations. The programs which must be accessed to support these facilities may exist scattered about the file system, or perhaps in a common directory with many other programs unrelated to configuration management tasks. The configuration management ADT can provide a coherent view of these operations and hide the organization or disorganization of the underlying programs.

6 X Window System Example

This section describes the use of abstract data types to interface to the X Window System. The X Window System[4], or simply X, defines a window system protocol, with which client applications and window system servers communicate. For application programmers to make use of X, a set of library routines (Xlib) and a set of higher level programmable interfaces (toolkits) are necessary. Ada interfaces to the Xlib routines and toolkits have been implemented that allow Ada applications to make use of X[8].

ACE contains a set of ADTs that provides interactive access to the X Window System routines. This permits ACE to be used as a rapid prototyping environment for the development of X applications. Interactively, windows may be created and destroyed, events processed and propagated, and graphical elements manipulated within windows.

The ACE ADTs for X support each window as a separately instantiated window object. Once a window object has been declared and elaborated, a set of operations may be applied to each window object. The window object and the applicable set of operations are defined through ACE ADTs. These ADTs consist of Ada package specifications and bodies. The majority of the package implementation consists of interfacing with the Ada binding to the X Window System.

The X Window System provides an excellent example of using ADTs to tailor the Ada command language to an individual or project's perspective. The X Window System is described and defined through a set of terminology that is based on window system concepts and the directions of the developers and implementors of X. X was designed to be hardware, operating system and language independent, hence the names of operations within the X library and the X toolkits are couched in window system terminology. This terminology may be flavored with the implementor's primary target environment dialect as necessary.

ACE defines two sets of X Window System ADTs within the standard ACE. Both of these support X in an Ada environment, but with different terminology. One set of X Window System ADTs is defined using X terminology; the other set of ADTs is defined in Ada terminology. An example of differences between X terminology and Ada terminology is in the operations used to instantiate a new window and to terminate an existing window. Using Ada terminology these operations are named *Create* and *Delete*, respectively. These names were chosen because they are used in Ada to define similar operations when applied to external files in Ada's file management packages. This supports the notion of a single paradigm to the user—Ada, where all operations, including those with other applications, are defined and invoked in a consistent, uniform manner. Using X terminology these operations are named *Create_Window* and *Destroy_Window*, respectively, to conform more closely to the standard names found in the X Window System. This permits users who are more familiar with X to operate in an environment in which they are more comfortable.²

²Note that these do not conform exactly to the C Xlib interface routine names of *XCreateWindow* and *XDestroyWindow*. Since ACE is an Ada environment, the X names were slightly modified to resemble typical Ada subprogram names. If the desire were to support a C-oriented interface to Xlib, an ADT could easily be defined that would support the C Xlib interface.

Each of these two abstractions are specified within ACE through ADTs. The user may then select which abstraction is most appropriate for the particular circumstance. Direct visibility of the desired set of operations may be acquired by issuing a "use" statement for the respective package.

7 Conclusion

The Ada Command Environment provides a command language that is Ada and supports an ADT view of the underlying operating system and application tools. Through an ADT definition, unique, heterogeneous systems may be presented to a user in a uniform and consistent description. The specifications of the ADT remain constant across heterogeneous environments (machines and operating systems), with different implementation of the ADTs to accommodate environmental differences. The ADT approach encompasses the paradigms of the Ada language and extends the programming language approach into the user's typical interactive environment.

References

- [1] Grady Booch. *Software Engineering with Ada*. Benjamin/Cummings Publishing Co., Menlo Park, California, 1987. (Second Edition).
- [2] David W. Embly and Scott N. Woodfield. *Assessing the Quality of Abstract Data Types Written in Ada*. Technical Report, Brigham Young University, Provo, UT, September 1987. BYU-CS-87-10.
- [3] William P. Loftus, Charles L. Oci, and John A. Thalhimer. 'The Ada Command Environment—ACE. In *Proceedings of Ada Expo '88*, Anaheim, California, October 1988.
- [4] Robert W. Scheifler and Jim Gettys. 'The X Window System. In *ACM Transactions on Graphics*, April 1986.
- [5] *Alsys Ada Sun Workstation Compiler User's Guide*. Alsys Inc., Waltham, Massachusetts, 1987.
- [6] *VADS Users Guide*. VERDIX Corporation, Chantilly, Virginia, 1987.
- [7] Reference Manual for the Ada Programming Language. United States Department of Defense, 17 February 1983. (ANSI/MIL-STD-1815A).
- [8] *Statement of Work for Ada-X Binding*. Science Applications International Corporation, San Diego, California, September 1987. STARS Foundation contract #N00014-87-C-0742.

8 Biography



John A. Thalhamer is the manager of STARS Foundations projects at Unisys. He has over nine years experience in the design and implementation of software support tools. His primary interests are in the areas of test/validation tools, Ada, compilers, and user interfaces. He holds an M.S. in Computer Science from Cornell University and a B.S. in Computer Science from the Pennsylvania State University. He is a member of the ACM and the IEEE Computer Society.



Charles L. Oei is a member of the Ada Command Environment development team. He has over six years experience in the design and development of compilation, windowing, and operating systems software. His primary interests are in the areas of software engineering, compiler technology, windowing systems, and engineering workstation technology. He holds an M.S. in Computer Science from the University of Illinois, and a B.S. in Physics from Butler University. He is a member of the ACM, and the IEEE and IEEE Computer Society.



William P. Loftus is the chief programmer for the Ada Command Environment STARS Foundation project. He has over five years experience in the design and development of compiler generation systems. His primary interests are in the areas of attribute grammars, compiler construction environments, and Ada. He holds a B.S. in Computer Science from Villanova University, and is a member of the ACM including SIGAda, SIGPLAN, SIGSOFT, and SIGART.



Ralph A. Foy is a member of the Ada Command Environment development team. He has over two years experience with software development environments and database management systems. His primary interest is in the area of software engineering environments. He holds a B.S. in Computer Science from Villanova University.

Authors' present address:

Unisys - Paoli Research Center
P.O. Box 517
Paoli, PA 19301

A Software Engineering Documentation Environment

Thomas J. Wheeler

U.S. Army CECOM
Center for Software Engineering
Ft. Monmouth N.J.

Abstract: Documentation size usually exceeds program size in most systems but the effort to understand and organize the technology and techniques of documentation and of providing tools for its preparation and its use is small compared to that applied to programming. This paper describes an effort to develop an innovative approach to the entry, storage, manipulation, and presentation of structured text, graphics, and formal (both specification and programs) documentation which enhances the efficiency and effectiveness of the authors and users of the documentation.

Our approach to the creation of the documentation environment provides a framework based on three concepts: the documentation is *structured*, and therefore some of the semantics can be captured, manipulated by the computer, and used to assist the creator and users of the documentation in their tasks; the structure of the documentation can be captured in a *formal specification*, allowing significant parts of the system to be methodically or even automatically generated; and the documentation environment has an *architecture* which centers around a semantic database of typed objects, relationships among objects, and semantic functions defining properties of objects. The database and applications programs are generated from formal specifications of the classes of objects, yielding a set of Ada packages. This paper describes the documentation environment's architecture, explores its use, and illustrates the technique of generating the database and applications. A companion paper describes the development of one of the applications and experience with the use of the technique.

Introduction

While documentation usually far exceeds program code in size in most systems, the effort to understand and organize the technology and techniques of documentation and of providing tools for its preparation and its use is miniscule compared to that applied to programming. This paper describes an R&D effort to develop technology for preparation, maintenance and use of textual, graphical and formal documentation in an integrated fashion within an advanced Software Engineering Environment. The paper describes an innovative approach to the entry, storage, manipulation, and presentation of *structured text*, graphics, and formal (both specification and programs) documentation which enhances the efficiency and effectiveness of the authors and users of the documentation.

The approach taken here to the creation of the documentation environment (a documentation environment is analogous to the combination of VI, TROFF, PIC, MS,... of Unix™) provides a framework for its construction based on three concepts: the documentation is *structured*, and therefore some of the semantics can be captured, manipulated by

the computer, and used to assist the creator and users of the documentation in their tasks; and the structure of the documentation can be captured in a *formal specification*, allowing significant parts of the system to be methodically or even automatically generated; and the documentation environment has an *architecture* for which centers around a semantic database of typed objects, relationships among objects, and semantic functions defining properties of objects. The database is generated from a formal specification of the classes of objects, yielding a set of abstract data type packages, in Ada, whose types are instantiated to form the database. Activities using the database are methodically developed as applications programs using the specifications, with parts of them generated from the specifications enhanced by semantic functions.

This paper first discusses the concept of structured documentation then describes the documentation environment's architecture, explore its use, and illustrate the technique of generating the database and applications. A companion paper describes the development of one of the applications and experience with the use of the technique.

Structured Documentation

When a document is written, the author has in mind an organizational pattern of the contents along with the standard organizing structure of the type of document being written. Readers, likewise, look for both typographic and semantic organizational patterns which are either implicit in the flow of the material or explicit, providing cues[Mey85] to the author's intended organization of the material. Documentation has structure, a significant part of a reader's ability to understand a document is the ability to discern the author's intended structure and then to understand the contents of the document within that framework[Jones85].

Everyone knows that documents have structure, the last paragraph of the introduction of this paper, along with its section headings tries to expose this paper's structure[Mey85], so the issue is not whether documents have structure but rather whether the documentation development environment should have knowledge of that structure and if so how does it get that knowledge and what use can it make of it.

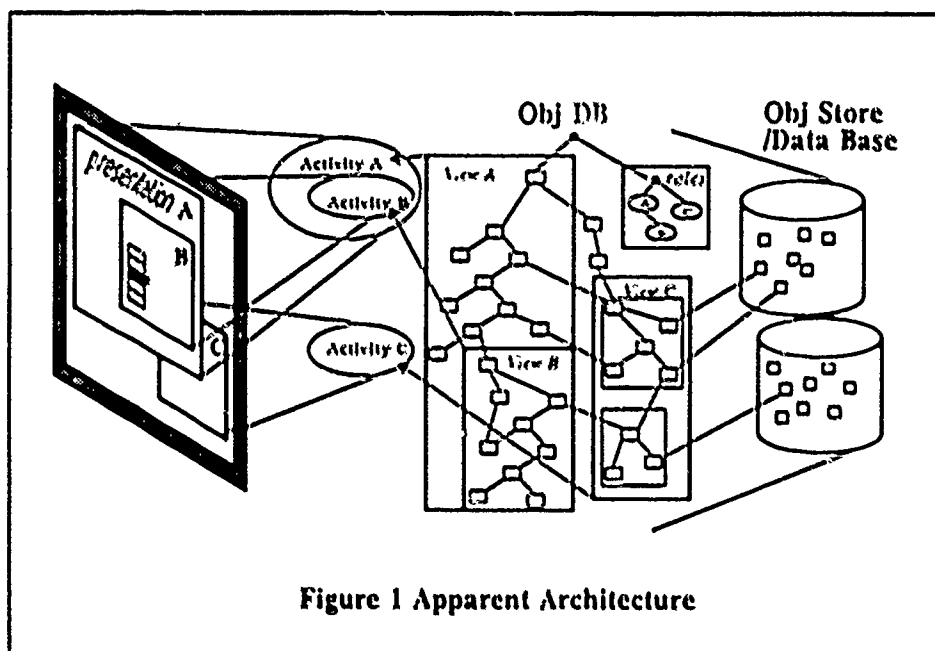


Figure 1 Apparent Architecture

The idea of using the structure of a document to assist in the automation of the document's preparation began with the Scribe[*K&A*] system which separated the work of the author from that of the document's designer. Scribe provided the author with structure describing commands to use to delineate the structural parts of the document; these commands were interspersed with the text in the input to the Scribe system, which was prepared with a text editor. More recently, the LaTeX [*L&A*] system has provided a similar capability as a macro package to the TeX[*K&A*] document preparation system and SGML [*SGML*] has defined a standard external transmission form for electronically transmitting and printing documents using similar concepts. All of these systems are "batch" systems producing the printed output document directly, and completely, from the textual input. They allow the author to encode the structure of the document in the text but they do not capture that structure in a form which allows processing other than printing.

The Etude[*ET&A*] system, and its decendent Interleaf[*WP&A*], provided interactive editing-formatting based on this structural paradigm by integrating an interactive formatting system into an editing system. The Interleaf system, on which this paper was created, provides a well engineered "what you see is what you get" (WYSIWYG), interface providing the structuring commands in an unobtrusive manner by function keys and menu selections. While all of these systems, progressively, provide authors with easy to use document preparation systems, they do not provide the integration of information storage and processing nor the semantic assistance which is provided for programming by the best integrated programming environments, eg. smalltalk and interlisp. That is, they do

not capture the structure in a way which allows the storage system to provide access to that structure to other tools for other purposes than printing.

In order to provide this integration and semantic support, we must look to the technology areas where high quality systems with these characteristics have been developed. Concepts and facilities to provide the integration we need have been developed in the database field and thus we base our architecture on the concept of an integrated documentation database. The major advances in the area of semantic support for automation have been made in compiler research and thus we base our approach on the use of the structure and techniques of compilers. The architecture and its implementation based on these techniques from database and compiler technology are discussed in the rest of the paper. The architecture is discussed in the next section, followed by a description and analysis of the usage of this type of environment along with the advanced functionalities made possible by this approach. A description of the implementation approach will follow that and then an example will illustrate the method and show its advantages.

Documentation Environment Architecture

The overall structure of the environment (Figure 1) is that of a database system[*DB&A*], a central database of structured *objects*, which are of numerous *types*, worked on by people in various *roles*, performing appropriate *activities*, creating, manipulating, and using collections of objects and their components, through *presentations* of those objects made available through appropriate *views* on the database. The types of objects important for this paper are various types of documents, but we expect the docu-

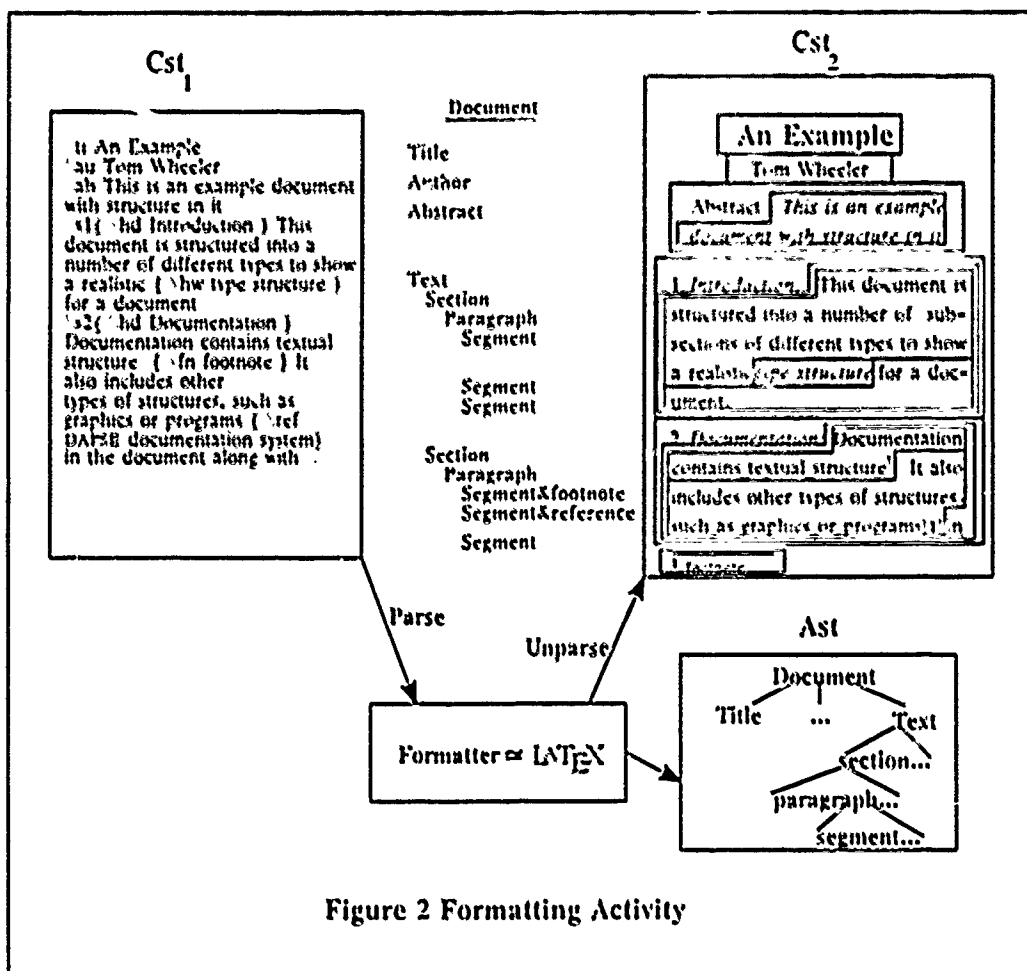


Figure 2 Formatting Activity

ments to be just part of an integrated collection of objects in the environment, to include programs, test plans, formal specifications, etc. in an overall system database.

At its core, the architecture provides, a repository of objects, along with their properties and relationships to other objects; and, as the objects may be structured, it provides for composition of objects from sub-objects along with their properties and relationships. Collections of objects, with subsets of their properties and relationships, are presented to users through views which provide contexts for those users to work. The users are involved in some activities using these views, with a user behaving in a role, known to the environment, with respect to each activity.

The database appears to the users as a, basically tree structured, collection of objects, defined by a set of abstract data types. These abstract data types define the abstractions provided by the views of the database; the implementations of the abstract data types provide the representations of the objects, in a form chosen and optimised by the environment's administrator (a role analogous to that of a database administrator). The contents of objects are

displayed within windows on the workstation screen and in hard copy form through presentations of the abstractions provided by the views.

The objects are structured (with the overall tree structure being provided by the "is made up of" relation of an object to its components) with the components of objects being themselves objects, thus there is no real distinction between the coarse grained structure used to describe the architecture and the fine grained structure of an object visible to an activity. This uniformity of concept provides a uniformity in the user's interface, as well as in the user's conceptualization of the system; the uniformity assists the user at both the surface (syntactic) and the deep (semantic) level. Making use of this uniformity, a preview of the main example to be presented later will illustrate the architecture.

The formatting activity (Figure 2) produces typeset text from textual input annotated by formal markup [sgml], in this case using the LaTeX markup language. The input is one of the presentations of the document being processed, being presented and edited in the LaTeX language. It is interpreted by the environment as a

concrete syntax tree structured by the LaTeX concrete grammar. It is transformed into the abstract syntax tree stored in the database and that is then transformed into the typeset document (another concrete syntax tree defined by the document's design which is encoded in the design's concrete grammar). The document's logical structure is shown in the figure as the indented list between the presentations of the two concrete syntax trees. The normal method of use, as shown by this example, is the manipulation and use of presentations derived from the abstract structured object(s) stored in the database.

Implementation of this architecture rests on a collection of specifications of the abstractions, their representation, and their presentations. Abstractions are specified through abstract grammars, presentations through concrete grammars and representations through data structures. These specifications are processed into abstract data type definitions, transformations of objects from their abstract type to or from one of the concrete types, and implementations of the abstract and concrete types, through a methodical technique developed in the compiling community and described in the section on Formal Specification and Software Generation.

Usage of the documentation environment, structured according to this database architecture will be covered in the next section, showing some of the advantages and innovations of this approach. The methodical implementation technique will then be described, showing how the database and its activities can be specified and the software implementing it can be methodically, or even automatically, derived from those specifications. After that, example activities, providing the formatting of structured text, will be described, illustrating the approach.

Documentation Environment Usage

The database orientation of the environment provides the well known advantages of information sharing, organization and control, but the enhanced concept used here also provides for assistance in performing work on a system's development, and functionality just not possible in non-automated systems. We will illustrate the capabilities of the environment¹ by showing the development and use of parts of an example of the developer's documentation of an embedded computer system[stop]. We will first show the information entry and storage, illustrating the information organization capabilities of the system and showing how this approach provides assistance to the user in performing necessary functions. We then illustrate the advanced functionality made possible by this approach by making use of the fine structure of the documentation and relationships among components of that structure to provide a hyper-media browsing capability for the documentation.

¹ As this is a technology development project, some parts exist in a finished form, some parts are in a prototype form and some parts only exist in conceptual design form.

The example we will use is the developer's documentation of a small embedded computer system using a style adapted by us[wait] from that used on the A-7 project at NRL[wait]. The hard copy form of this example is available in Ada Letters[wait, wait]. The entry of the contents of the documentation is done through the use of a formatting language or a structure editing system generated from a structural specification of the logical structure of the documentation; we will illustrate both.

Text entry, using the formatting language, takes textual input interspersed with logical markup commands, for example, the beginning of the behavior section of the system specification would be input as:

```
\sec{Stoplight Control System Behavior Subsystem}
\em{}

\sub{Conceptual Model}

\par This Stoplight Control Software Subsystem Specification uses a conceptual model of the behavior, a state machine, for motivation in understanding the functioning of the system and as a framework to describe the functioning of the system. The stoplight control system can be understood as a finite state machine whose states are the combined states of an approach for each direction, each of which can be empty or occupied, and a light which can be red, yellow or green in each direction. The traffic flow in the intersection controls the values of the approach states and the system adjusts the light states to control the traffic.

\sub{System Behavior}

\par The behavior of the Control Software is specified as a \em{finite state machine} in terms of the following concepts:

\inp{There are two \kw{Directions}, named \em{N_S} and \em{E_W}
An \kw{Approach} for each \kw{direction} which may be \em{Occupied} or \em{Empty}.
A \kw{Light} for each \kw{direction} which may be
\em{Red}, \em{Yellow} or \em{Green}.
A \kw{Switch} labeled \em{Off}, \em{On} or \em{Not_Working}.)}
```

Given the meaning of the commands, (\sec = section, \sub = subsection, \par = paragraph, \inp = indented paragraph, \kw = keyword, \em = emphasized text) this input would be parsed to place the contents into a structured object in the database, as will be shown in the next section, and that object could then be formatted to produce the following output:

4. Stoplight Control System Behavior Subsystem.

4.1 Conceptual Model.

This Stoplight Control Software Subsystem Specification uses a conceptual model of the behavior, a state machine, for motivation in understanding the functioning of the system and as a framework to describe the functioning of the system. The stoplight control system can be understood as a finite state machine whose states are the combined states of an approach for each direction, each of which can be empty or occupied, and a light which can be red, yellow or green in each direction. The traffic flow in the intersection controls the values of the approach states and the system adjusts the light states to control the traffic.

4.2 System Behavior.

The behavior of the Control Software is specified as a *finite state machine* in terms of the following concepts:

- There are two Directions,
named *N_S* and *E_W*
- An Approach for each direction
which may be *Occupied* or *Empty*.
- A Light for each direction which may be
Red, *Yellow* or *Green*.
- A Switch labeled *Off*, *On* or *Not_Working*.

Which has the same logical structure as the input, with the concrete syntax considerably (and *usefully*) different.

Structure editors have a knowledge of the logical structure of the object(s) they are editing. This knowledge is provided to the editor in the form of a specification and is presented to the user in the form of help or assistance in the entering and manipulating of the contents of the object. We will illustrate this by showing the use of a form based reference editor for entering bibliographic references in the text. For instance, when attaching a bibliographic reference to a segment of text, a window with a form based editor would appear on the user's screen; it would request the type of reference (journal paper, book, etc.) and then prompt for the contents:

Reference:	
Type:	book
Author:	<input type="text"/>
Title:	<input type="text"/>
Publisher:	<input type="text"/>
Date:	<input type="text"/>

When filled in, it would enter the information in the bibliographic database and attach a reference relation to the text segment.

Formal Specification - Software Generation

The idea of methodically, even automatically, generating programs from specifications started with the "structured programming" methods of Jackson^[2,4] and Wirth^[4,6] in the mid seventies. Here, one would define the data structures which a program needed to work and, from the natural correspondence between data structures and program structure (eg arrays correspond to loops, records correspond to blocks) the program's structure would be methodically derived. The automatic generation of both data structures and program structure, in the limited domain of programming language compilers, came into being with the compiler-compilers, at about the same time^[5,6,7]. The extension of this technique to programming environments occurred later with the Gandalf^[8] project at CMU and the DAPSE project^[9]. We are extending the DAPSE approach into the (structured) documentation area.

The specifications are written in an attribute grammar^[10] notation and a combination of methodical and automatic generation of the database and programs is used to construct the environment facilities (methodical generation is used to prototype a function, working out the details so that an automatic tool can be generated).

The technique is best explained by illustration: suppose it is desired to make a system which formats text, including subscripts, i.e. an input of *E_{sub1}.val* should produce *E₁.val*. Using this technique, we would write an attribute grammar¹, specifying the structure and functionality of the system:

```
S ::= B;
    (B.ps := 10)
    (S.ht := B.ht)
B ::= B1 B2;
    (B1.ps := B.ps)
    (B2.ps := B.ps)
    (B.ht := max(B1.ht, B2.ht))
B ::= B1 sub B2
    (B1.ps := B.ps)
    (B2.ps := shrink(B.ps))
    (B.ht := disp(B1.ht, B2.ht))
B ::= text
    (B.ht := text.ht X B.ps)
```

Where *S* = segment, *B* = block, *ps* = point size, *ht* = height, *shrink* = function to calculate the size of a numeral when used as a subscript, *disp* = function to calculate the height of two boxes displaced vertically, and (*attr := val*) is an attribute semantic function

¹ We limit the grammar here to specifying the height calculation for simplicity

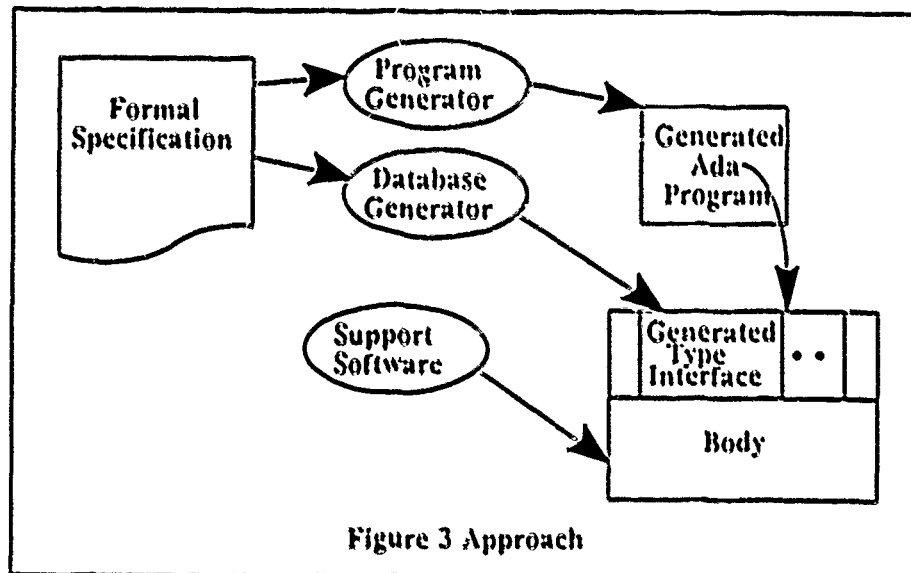


Figure 3 Approach

This grammar is methodically transformed into a storage definition module (note that *B* is defined by three productions *B* ::= *DB*, *B* ::= *BsubB*, and *B* ::= *text*) which contains the following data types to store Block nodes:

```

type B_production_variety is (BisDB, BisBsubB, BisText);
type B_node_contents (Variety: B_production_variety := BisText);
type B_node is access B_node_contents;
type B_node_contents (Variety: B_production_variety := BisText)
is record
    ht: height;      -- synthesized attribute
    ps: point_size;  -- inherited attribute
    case Variety is
        when BisDB => B1, B2: B_node; --B: = DB
        when BisBsubB => B1, B2: B_node; --B: = BsubB
        when BisText => text: string; --B: = text
        text_ht: height; --intrinsic
    end case;
end record;

```

and a function module which provides the following function to process Block nodes:

```

function B (Bn: B_node, Bn_ps: point_size) return height is
begin
    case Bn.Variety is
        when BisDB => Bn.B1.ps := Bn_ps; -- set inherited
            Bn.B2.ps := Bn_ps; -- attribute of children
            Bn.ht := max( B(Bn.B1, Bn.B1.ps),
                B(Bn.B2, Bn.B2.ps ));
            -- calculate attribute(s) of self
        when BisBsubB => Bn.B1.ps := Bn_ps;
            Bn.B2.ps := shrink( Bn_ps );
            Bn.ht := disp( B(Bn.B1, Bn.B1.ps),
                B(Bn.B2, Bn.B2.ps ));
        when BisText => Bn.ht := Bn_ps * Bn.text_ht;
    end case;
    return ( Bn.ht ); -- return synthesized attribute(s) of self
end B;

```

Automatic generation of software proceeds similarly, with top down generators producing recursive data structures and functions similar to those illustrated above, definite clause grammars in PROLOG[10], or tables similar to those used in [W77a]. Bottom up generators using table driven techniques similar to those used in [YACC]. The approach to developing systems using this technique is illustrated in figure 3.

Example

We will illustrate the documentation environment and its construction by developing the structure of a component of the environment, the subsystem which transforms textual input into a structured document in the database and transforms that document into typeset output. This function can be viewed as a batch formatter as illustrated in figure 4, but in our approach it is viewed as the three components that were illustrated in figure 2: a translator from the textual concrete syntax to the abstract syntax, a database to store the abstract syntax, and a translator from the abstract syntax to the output concrete syntax.

The database to store the abstract syntax is generated from a specification of the abstract structure of a document, as is shown on the right hand side of figure 5, which specifies the productions showing the components of the objects which make up a document along with their attributes (in italics). This generates the type definition modules out of which the database is constructed by the parsing function; the database resulting from inputting the example document is shown on the left side of figure 5. The parsing function is generated from a concrete grammar, such as:

```

Document ::= (Title, Author, Abstract, Text, References)
Title ::= "\ti" Text
Author ::= "\au" Text
Abstract ::= "\ab" Text
etc

```

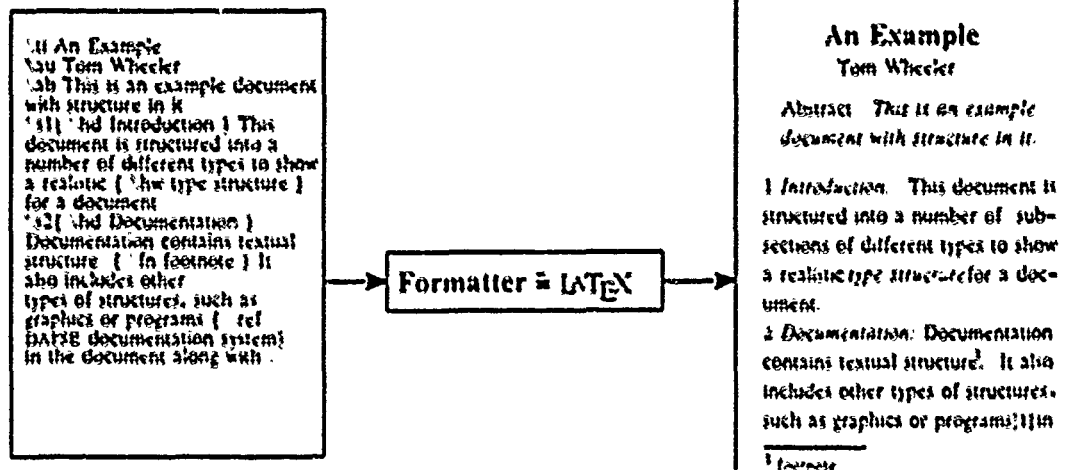
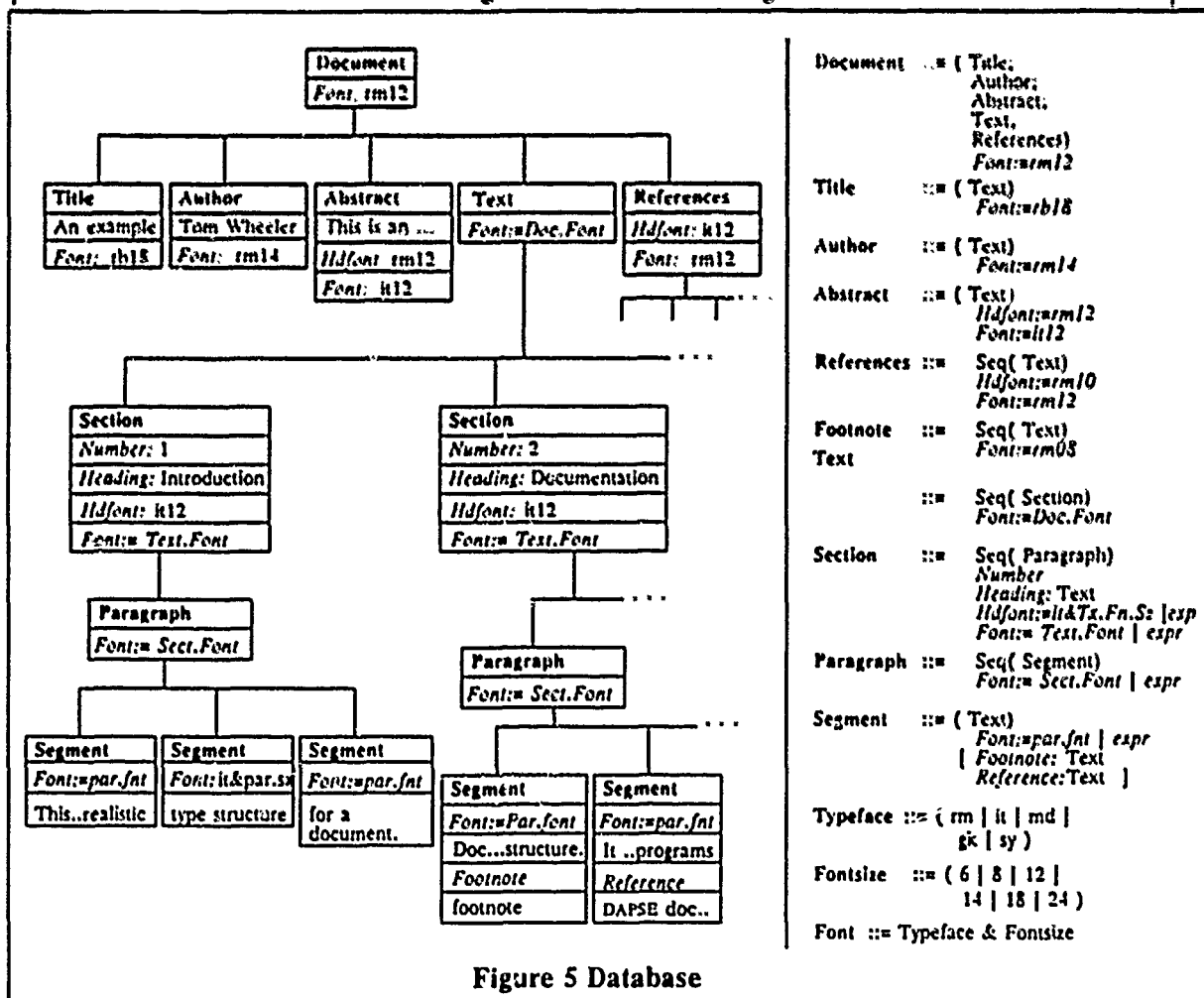


Figure 4 Text Formatting



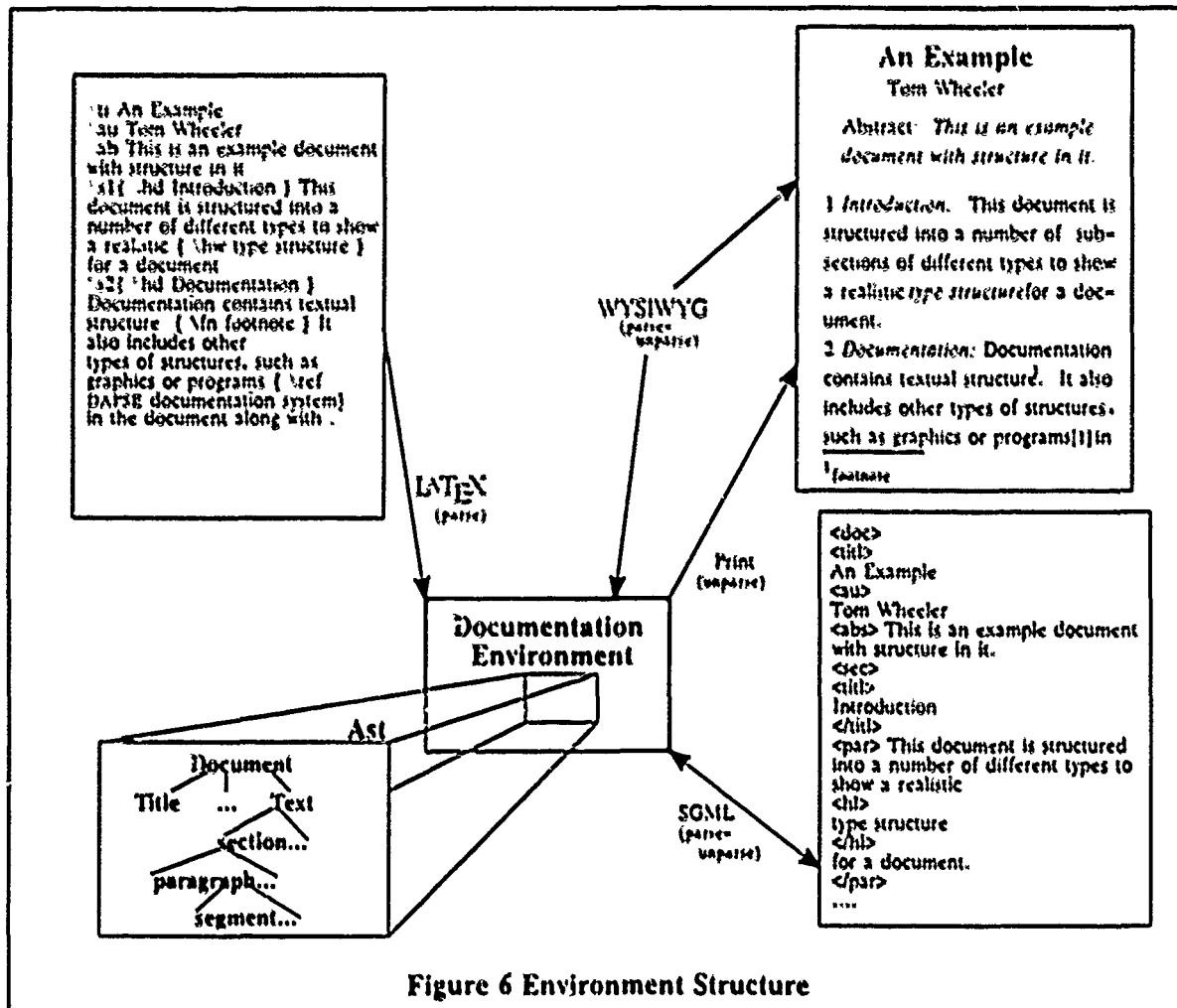


Figure 6 Environment Structure

along with the abstract grammar producing, for example, recursive procedures like:

procedure DOCUMENT (D: out Document_Structure) is
begin

```

  if Next_Token = "<ti>" then
    TITLE( T); -- reads title string into T
  elsif Next_Token = "<au>" then
    AUTHOR( A); -- reads author string into A
    ...
  end if;
  D := Make_Document_Node( T, A, ...); -- uses
    -- abstract grammar to construct node
end DOCUMENT;
```

when the recursive decent generation technique is used. The unparser is generated similarly, walking the tree like the function "B" above rather than scanning the input like "DOCUMENT" does.

The main advantages of this architecture are the opportunities for reuse of the subfunctions which become manageable entities in the architecture, the productivity

advantages of the software generation approach fostered by the architecture, and the opportunities for providing innovative functionalities made possible by it. As an example of the reuse and productivity advantages, a SGML exporter is a, relatively simple, tree walking unparser which can be generated from the ast grammar specification and a SGML concrete grammar specification, figure 6, while a SGML importer is a parser similar to the LaTeX parser. As an example of the provision of innovative functionality, consider a WYSIWYG document entry system, in this architecture it is a combination of an unparser, similar to the typesetting unparser, along with a parser of that concrete syntax, figure 6. A final example of innovative functionality is a hyper-text browser which provides a menu entry for each relation from a node, follows that relation to its destination node upon selection and displays the destination node using the WYSIWYG's unparser.

Conclusions

The capturing of the structure of the documentation in the documentation environment presents many advantages, in the providing of advanced functionality, in the flexibility of system and in the productivity of the development and system evolution process. The documentation environment project introduced here has shown that the architecture and approach described are feasible and desirable, leading to high quality high functionality systems.

References

- [CIS7] Clocksin, W. & Mellish, C. *Programming in PROLOG* Springer-Verlag, Berlin 1987
- [DAPSE] Marcus, M., Seahsner, S., Satley, K., & Albert, E. "DAPSE: A Distributed Ada Programming Support Environment" 2nd IEEE Ada Applications & Environments Conf. 1986
- [Gandalf85] The Gandalf Project, *Journal of Systems and Software*, May 85(entire issue)
- [Ham81] Hammer, M. "Etude: an Integrated Document Processing System" 1981 Office Automation Conference
- [Ja75] Michael Jackson *Principles of Program Design* Academic Press London 1975
- [Jon85] David Jonassen *The Technology of Text* Educational Technology Publications, Englewood Cliffs NJ 1985
- [Jon85a] Jonassen, David "Generative learning vs Mathemagenic Control of Text Processing" in [Jon85]
- [Kn84] Donald E. Knuth *The TeXbook* Addison-Wesley, Reading MA 1984
- [Kn68] Knuth, D. "Semantics of Context Free Languages" *Mathematical Systems Theory* 2(2) 1968
- [La86] Leslie Lamport *LaTeX User's Guide & Reference Manual* Addison-Wesley, Reading MA 1986
- [Mey85] Meyer, Bonnie "Signaling the Structure of Text" in [Jon85]
- [NRL] "The A-7 Documents" available from NRL code 7590 Wash., D.C. 20375
- [ReS4] Brian Reid *Scribe Document Production System* Unilogic, Ltd. 1984
- [SGML] *Standard Generalized Markup Language* ISO 8879
- [wp85] Workstation Publishing System, Interleaf Inc. 1985
- [Wh81] T. Wheeler "Embedded System Design with Ada® as a System Design Language" *The Journal of Systems and Software*, Vol 2 pp.11-21 1981.
- [Wh86] Wheeler, T. "An Example of the Developer's Documentation for an Embedded Computer System Written in Ada" *Ada letters* VI-no.6 1986
- [Wh87] Wheeler, T. "An Example of the Developer's Documentation for an Embedded Computer System Written in Ada" Part II *Ada letters* VII-no.1 1987
- [Wir76] Niklaus Wirth *Algorithms + Data Structures = Programs* Prentice-Hall, Englewood Cliffs NJ 1976
- [YACC] Johnson, S. "Yacc-yet another compiler compiler" *Bell Labs Tech. Report #32* 1975

Biography



Dr. Tom Wheeler is in charge of the research program in CIECOM's Advanced Software Technology Division. He has the Ph.D. Degree in Computer Science from Stevens Institute of Technology, the Masters degree in CS from Fairleigh Dickenson University, the BSEE degree from Monmouth College, and the BA degree in Physics from La Salle University. He has worked in Computer Aided Design, Software Methodologies, and Software Environments. He has performed Research in formal methods for software development, system's programming, system design, distributed programming, requirements development, and rapid prototyping. He has taught Ada, computer science, and software engineering at Stevens Institute of Technology, Monmouth College, in industry and within the Government. He is currently organizing a research project into an alternate approach to Software Engineering Environments and Information Systems.

DOCUMENTATION GENERATION SYSTEM

Dudley Rodericks, Ismael Rivera, Bruce Kolofsky, Roberto Quinones

U. S. Army Communications and Electronics Command
Center for Software Engineering
Fort Monmouth, New Jersey

ABSTRACT

The goal of this project was to investigate the application of compiler technology to the development of an integrated documentation generation system. A prototype text formatter was developed as a demonstration of the application of this technology to such a system.

1.0 INTRODUCTION

Once a system has been designed and developed, it is often the supporting documentation that is of the greatest concern to the developer. The required documentation is very often neglected due to schedule overruns. An integrated environment with tools to support the user would be a very desirable system. As such, it was decided to design and implement a text formatter using aspects of the existing compiler technology. This text formatter was treated as a prototype subsystem of the overall envisioned system and was attempted mainly to investigate the application of this technology to the area of documentation engineering.

The text formatter was designed to be a batch system and acts mainly as a filter. From the user's point of view, the system reads in an input file, acts on it (as indicated by the appropriate commands contained within the document), and produces some output. An ASCII editor is needed to enter the text in a file. Commands are to be used to achieve the desired

format. These commands would be provided in a User's Manual.

Since this project was to investigate the application of compiler technology, a context-free grammar was first defined. An intermediate representation, the Abstract Syntax Tree, was then decided upon. Following this, attributes were chosen and the scanner and parser designed. These are discussed in the following sections. It must be noted that the overall system is not restricted to a formatter. Once the intermediate representation is known, several applications may be developed based on the structure of this representation. This may be done by producing different back-end code generators. As such, several subsystems may be developed and integrated as part of an overall environment.

2.0 COMPILER DESIGN

This section provides some background on the aspects of compiler structure that were deemed to be of importance to this project. The main phases of a compiler (4) are the following :

(a) **Scanner** : The function of the scanner is to read in characters from a source file and produce tokens as an output. These tokens are defined by the context-free grammar.

(b) **Parser** : The parser reads in the tokens produced by the scanner and sorts them

into groups based on the productions defined in the context-free grammar.

(c) **Semantic Routines** : Semantic routines check the semantics of the constructs and perform a translation by producing an intermediate representation.

(d) **Code Generator** : The intermediate representation produced by the semantic routines is then converted into some target machine code. As such, if code were needed for a different target machine, only the back-end code generator would have to be changed.

This, very briefly, describes the structure of a compiler and is shown in Figure 1.

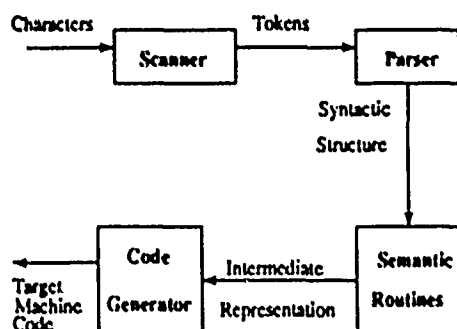


Figure 1 Structure of a Syntactic Compiler

3.0 CONTEXT-FREE GRAMMAR

A context-free grammar was first defined for this project as this would influence the design of the parser. An LL(1) type grammar was decided upon as this would facilitate top-down parsing. This grammar utilized the idea of productions or rules for the system followed by terminals (the right-hand side of a production)

and non-terminals (the left-hand side of a production). All non-terminals are first resolved into terminals by the system before proceeding and this is done as defined by the appropriate production. As such, by knowing the productions and recognizing a valid non-terminal, the system knows what should follow. This grammar is described as follows :

```

<Document> --> <Title> <Date>
               <Author> <Sections>

<Title>       --> <Line> ( <Line> )

<Line>        --> Char_String

<Date>        --> Char_String

<Author>      --> <Line> ( <Line> )

<Sections>    --> <Sec_Title>
                  ( <Paragraphs> )
                  ( <SubSections> )

<Sec_Title>   --> Char_String

<Paragraphs> --> <Block> ( <Block> )

<SubSections> --> <SubSec_Title>
                  ( <Paragraphs> )

<SubSec_Title> --> Char_String

<Block>       --> Char_String
  
```

The < > brackets indicate a non-terminal which must be resolved into terminals. The braces () denote optional items from 0 onwards. Associated with the grammar defined above are attributes. These attributes are associated with certain terminals and non-terminals. These are shown below :

ITEM	ATTRIBUTE
Paragraphs	Indent of three spaces

Section_Title	Section_No (Generated by system)
SubSec_Title	SubSec_Title_No (Generated by system)
Char_String	Font, Size, Quality, Underline, Bold

Appropriate default settings are provided for the attributes.

4.0 SYSTEM DESIGN

The design of the system consisted of the Scanner, the Parser, and the Code Generator. This was consistent with the effort to explore compiler technology as a development technique for the system.

Scanner : The Scanner follows the same functionality as the scanner in a compiler scheme. It consists of the Reader and the Translator. The reader reads in characters from the input file and the Translator converts these characters to tokens. These tokens are the terminals and appropriate commands.

Parser : The Parser consists of the Token Identifier, Token Translator, Attribute Manager, and the Tree Manager. The Token Identifier receives tokens from the Scanner and groups them into commands and attributes. These commands and attributes are then passed to the Token Translator. The Token Translator sends the attributes to the Attribute Manager and the commands to the Tree Manager. The Attribute Manager groups incoming attributes into sets and sends them to the Tree Manager. The Tree Manager receives commands and sets of attributes and creates the Abstract Syntax Tree with the appropriate nodes. It then enters the

accompanying text at the nodes. Each node has certain attributes associated with it. These attributes are either set by the user or are provided for by default settings within the code.

Code Generator : The Code Generator consists of the Tree Reader, Node Identifier, and the Node Translator. The Tree Reader reads the Abstract Syntax Tree and retrieves the nodes. The nodes are sent to the Node Identifier which decides what type of node they are. This node information is then passed to the Node Translator which produces the desired formatted output. The Code Generator is dependent upon the application desired. There are two generators for this project. One produces output targeted for an IBM compatible dot matrix printer while the other produces output in Postscript.

The system design is shown in Figure 2, and the first-level decomposition is shown in Figure 3.

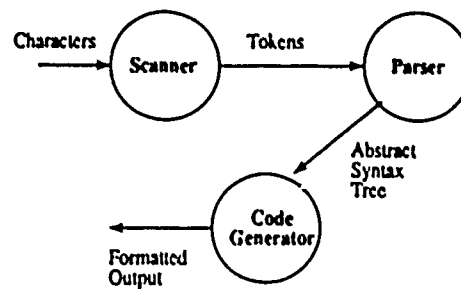


Figure 2 Top-Level System Design

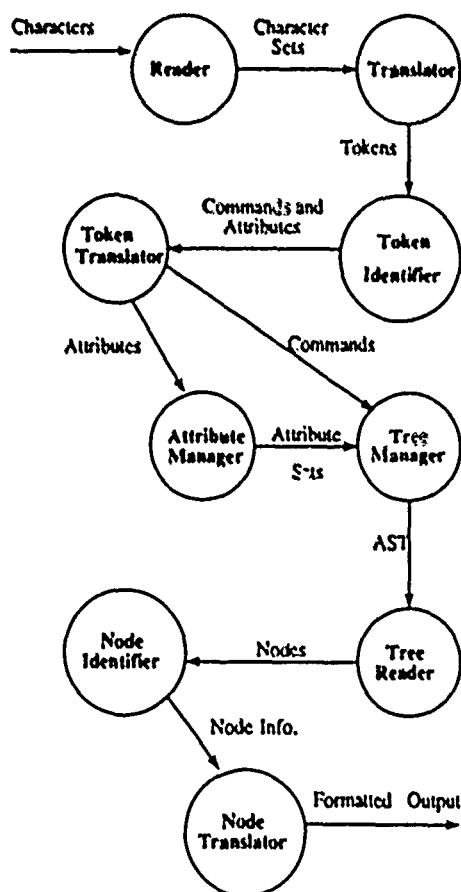


Figure 3 Breakdown of System Design

5.0 SOFTWARE DESIGN

The software design closely followed the system design. The language used to code this

system was Ada¹. The compilers used were the Alsys² Ada compiler on a Zenith³ AT and the Verdix⁴ Ada compiler on the Sun⁵ workstations. The modules produced are shown below :

Packages :

(a) Token_Scanner : This package encapsulates the functions of the Scanner from the system design. It consists of the procedure Open_File and the procedure Get_Token.

(b) Parser : This package addresses the functions of the Parser from the system design. It consists of the procedure Append_CST.

(c) Code_Generator : This package contains the procedure Generate_Output which provides the functions for the Code_Generator from the system design.

(d) Type_Declarations : This package contains all the data types necessary for the other packages.

(e) Tree : This package contains the types necessary for the Parser and Code_Generator packages in order to isolate these types from the Scanner, which does not need them.

Several types of nodes were needed in the Abstract Syntax Tree. The type of node was indicated by the structure of the context-free grammar (eg. Section nodes, Subsection nodes etc.). These nodes then had certain attributes associated with them. Discriminant records were used to represent the various kinds of nodes. The nodes were differentiated based on the contents of a field called Kind. The software top-level dependency diagram is shown in Figure 4.

1. Ada is a registered trademark of AIPO.

2. Alsys is a registered trademark of Alsys, Inc.

3. Zenith is a registered trademark of Zenith Data Systems.

4. Verdix is a registered trademark of the Verdix Corporation.

5. Sun is a registered trademark of Sun Microsystems.

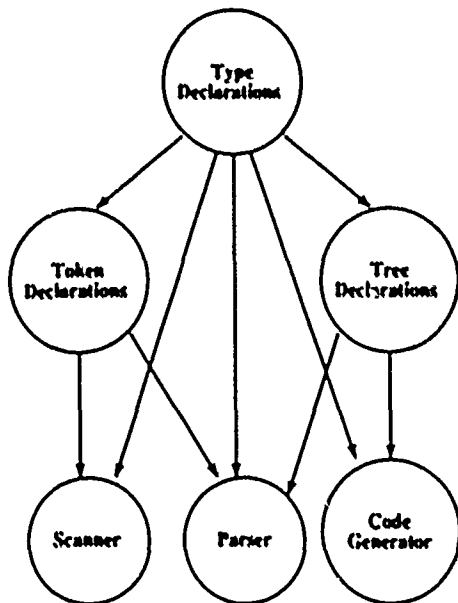


Figure 4 Ada Package Dependency Diagram

6.0 ENHANCEMENTS

It must be noted that the main purpose of this project was to demonstrate the use of a technology for a particular area. The system developed was largely intended to be a vehicle for the purposes of demonstration. If this prototype were to be further developed, the following enhancements should be considered :

(a) The system could be made interactive (WYSIWYG) and menu-driven. This would

make it more user-friendly. The ability to input the productions desired and have the scanning and parsing routines incorporate these rules would make for flexibility.

(b) An underlying database could be integrated. The Abstract Syntax Tree could then be permanently stored in the database instead of being resident in memory. This would allow for the incorporation of roles within the environment.

(c) The technique of swapping could be added in order to increase performance.

(d) The option for different output generators could be added. Selection could be made from a menu. Thus the user could choose to obtain a printed copy, observe the formatted copy on the screen, or achieve any goal for which there exists an output generator. This would not affect the parser or the scanner.

7.0 CONCLUSION

This project demonstrated the porting of compiler technology to a documentation system. The major achievement is the realization of the Abstract Syntax Tree. Once the structure of this tree is established, several tools may be developed and interfaced to provide a comprehensive environment.

This prototype was a tool that demonstrated this fact. This tool could be further refined into a viable commercial product. From a conceptual level, there were no actual hardware or software dependencies. If output were desired for a different target machine or purpose, then only the Code Generator modules need be changed. The sample commands used were omitted from this paper. It has been the intention of this paper to demonstrate the application of a technology to a certain area and not to involve the reader in the details of the prototype. The prototype has only been discussed to the level

which helps achieve this goal.

ACKNOWLEDGEMENTS

The authors wish to express their thanks to Dr. T. J. Wheeler for his technical guidance in this effort. The authors also wish to acknowledge the efforts of Jay Seo for his help in providing the necessary background material for this project, and Bruce Gray for his support and belief in the project and the project team.

REFERENCES

1. Ada Language Reference Manual.
2. Alslys Ada User's Guide.
3. Warren, Kickenson, and Snodgrass, A Tutorial Introduction to using IDL, Department of Computer Science, University of North Carolina.
4. Charles N. Fischer and Richard J. LeBlanc, Jr., "Crafting a Compiler", (Benjamin/Cummings), 1988.
5. Defense System Software Development (DOD-STD-2167).
6. Sal Gambino et. al., Text Formatter System, Stevens Institute of Technology, 1987.

ABOUT THE AUTHORS

Dudley Rodericks is an Electronics Engineer with the U.S. Army Communications-

Electronics Command, Center for Software Engineering, Advanced Software Technology, Fort Monmouth, New Jersey. He received his B.S. in Electrical Engineering from Texas A&I University at Kingsville, Texas, and his M.S. in Software Engineering from Monmouth College, New Jersey. He is currently working in the System Software Technology division.

Ismael Rivera is an Electronics Engineer with the U.S. Army Communications-Electronics Command, Center for Electronic Warfare/Reconnaissance, Surveillance, and Target Acquisition, Fort Monmouth, N.J. He received his B.S. in Computer Engineering from the University of Puerto Rico, Mayaguez, Puerto Rico, and his M.S. in Software Engineering from Monmouth College, New Jersey. He is currently working in the Technical Support division.

Bruce Kolofsky is an Electronics Engineer with the U.S. Air Force, Air Logistics Command, Software Production, Robins Air Force Base, Warner Robins, Georgia. He received his B.S. in Chemical Engineering from the University of Florida at Gainesville, Florida, and his M.S. in Software Engineering from Monmouth College, New Jersey. He is currently working in the Maintenance division.

Roberto Quinones is an Electronics Engineer with the U.S. Army Communications-Electronics Command, Center for Software Engineering, Tactical Systems, Fort Monmouth, New Jersey. He received his B.S. in Computer Engineering from the University of Puerto Rico, Mayaguez, Puerto Rico, and his M.S. in Software Engineering from Monmouth College, New Jersey. He is currently working in the Mobile Subscriber Equipment division.

REDUCING SOFTWARE DEVELOPMENT COSTS WITH ADA

Jeffrey R. Carter, Senior Engineer, Software

Martin Marietta Astronautics Group, Denver, Colorado

Abstract

Increases in the abstraction of languages have historically resulted in significant reductions in the cost of software development by eliminating or reducing software development phases. These changes in the software-development lifecycle have significantly reduced the cost of software development for identical problems. Ada is a significant increase in abstraction over other languages. Ada must change the traditional software-development lifecycle if it is to provide a significant reduction in software development costs. Comparing the results obtained using the Martin Marietta Ada Implementation Method with those obtained using traditional methods demonstrates that using Ada with a method which incorporates the software engineering mind set does change the lifecycle. This results in a significant reduction of software development effort and cost.

Introduction

Increases in the abstraction of languages have historically resulted in significant reductions in the cost of software development by eliminating or reducing software development phases. Obvious examples include using assembly languages instead of machine code and using higher-level languages, such as FORTRAN, instead of assembly languages. The former eliminated the phase in which machine code was determined, while the latter significantly reduced the coding phase. Perhaps less obvious is the use of high-level languages which contain a complete set of control structures, which eliminated the phase in which a "structured" program design language was translated into the implementation language. Changes in the software-development lifecycle such as these have significantly reduced the cost of software development for identical problems.

Ada is a significant increase in abstraction over other languages [1]. Ada has been shown to reduce the effort required to develop software when used as the implementation language with traditional software development methods [2].

However, Ada must change the traditional software-development lifecycle if it is to provide a significant reduction in software development costs. Many researchers continue to discuss Ada with the assumption that the full, traditional-lifecycle model is applicable. This assumption is not valid, as this paper demonstrates. If it were true, using Ada would not result in any major reductions in software costs.

In my work with the Martin Marietta Ada Implementation Method (MMAIM) [3, 4] I have used MMAIM on a number of problems which occur frequently in the literature and so have been worked with traditional-lifecycle methods for implementation in traditional languages. MMAIM reflects and enforces the modern software engineering "mind set." By comparing the results obtained using MMAIM with those obtained using traditional methods, I can demonstrate that using Ada with a method which incorporates the software engineering mind set does change the lifecycle. This results in a significant reduction of software development effort.

The Cruise-Control Problem

The problem of a cruise control for an automobile occurs in nearly all the real-time literature. Ward and Mellor [5] present such a problem, but it bears no resemblance to any actual cruise-control system I have encountered. In this paper I will use a version of the cruise-control problem to demonstrate the elimination of lifecycle phases using Ada and MMAIM.

The cruise-control system is intended to control the speed of a car by maintaining a constant speed. The controls for the system consist of a two-position switch labeled "off" and "on," two buttons labeled "set" and "resume," and a sensor connected to the brake pedal. These controls generate five unique interrupts. The off-on switch generates an interrupt whenever its position is changed; one interrupt is generated when the switch is moved to the "off" position and another when it is moved to the "on" position. The set and resume buttons each generate an interrupt when they are pressed. The brake sensor generates an interrupt when the brake pedal is depressed.

The off-on switch turns the system off and on. When the system is off it has no effect on the car's speed, and will only respond to the system being turned on. The system ignores the other controls. Note that the software continues to operate when the system is turned off.

When the system is on, it can be turned off, or it can be instructed to maintain the car's current speed by pressing the set button. The system ignores any other controls.

When the system is maintaining a speed, it can be turned off, or it can be interrupted by pressing the brake pedal. The driver can also override the system by pressing the accelerator pedal. In this case the driver can have the system maintain the new speed by pressing the set button. The system ignores the resume button.

When the system is interrupted, it can be turned off, or it can be set to maintain a new speed, or the resume button will cause the system to return the car to the previous speed and maintain it. The system ignores the brake pedal.

Note that this system has some simplifications from an actual cruise control. For example, there is no interaction with the car's transmission. Real cruise-control systems take the transmission into consideration. Also, we will assume that the specific computer for the system allows the software to ignore the interrupts which the controls generate. Few actual computers behave this way.

The Software-Development Process

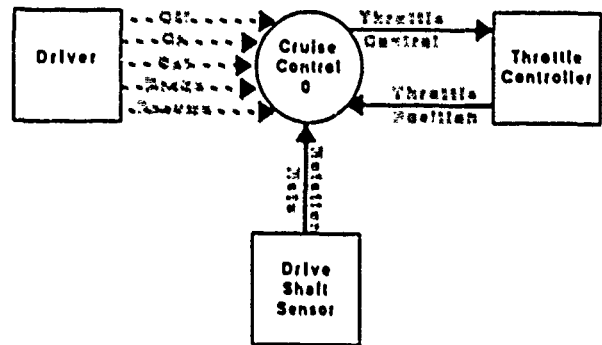
In this paper I am concerned only with the software-development process. Any system-level concerns, such as hardware-software partitioning, have already been done. I am not discussing what methods are appropriate for dealing with system-level concerns.

Traditional software-development methods begin with a phase which is usually called analysis, specification, or systems design. One method which is widely used for this software-development phase is real-time structured analysis. In this section of the paper I will work through the steps involved in applying real-time structured analysis to the cruise-control software-development problem, and compare this traditional approach to MMAIM. Both these methods produce both textual and graphical results. Since the graphical products contain more information, I will concentrate on them.

Step 1--Real-Time Structured Analysis

In applying real-time structured analysis to the cruise-control problem I will use Ward and Mellor's solution [5] as a guide. The first step in applying real-time structured analysis is to describe the context of the software. The context diagram shows the software as a circle and the things in its environment with which it interacts as boxes. Solid arrows represent data flows between the software and the boxes, and dashed

arrows represent control flows. Data flows are not associated with control flows. Working along the lines suggested by Ward and Mellor produces the context diagram in Figure 1.



Cruise Control Context Diagram
Figure 1.

Step 1--MMAIM

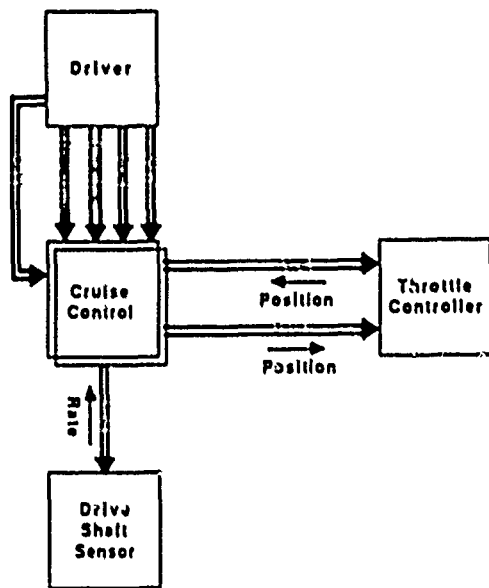
MMAIM's first step identifies the software boundary, the entities external to the software with which it interacts, and the interfaces across or through the software boundary. These are shown on the External Entity Graph, or EEG. Figure 2 presents the EEG for the cruise control. The EEG shows the software as a double-line box and the external entities as single-line boxes. The double-line arrows represent interfaces across the software boundary. The direction of the arrow represents the direction of control; we would say that the software "calls" the throttle controller. This means that the software decides or controls when it will get or change the throttle's position. The small arrows associated with the interfaces represent data flows associated with the interface.

Step 1--Comparison

The first steps of the two methods are very similar. MMAIM provides some information not given by real-time structured analysis: the association of data with control.

Step 2--Real-Time Structured Analysis

Real-time structured analysis' next step decomposes the software into its major functions, data stores, and the data and control flows between them. Figure 3 is the top-level data flow diagram for the cruise control. The solid-line circles represent data-transforming functions. The dashed-line circles represent control-transforming functions. Two parallel horizontal lines represent data stores, which usually mean global or common variables. Note that, although this diagram inherits arrows from the context diagram, a single arrow on the context diagram



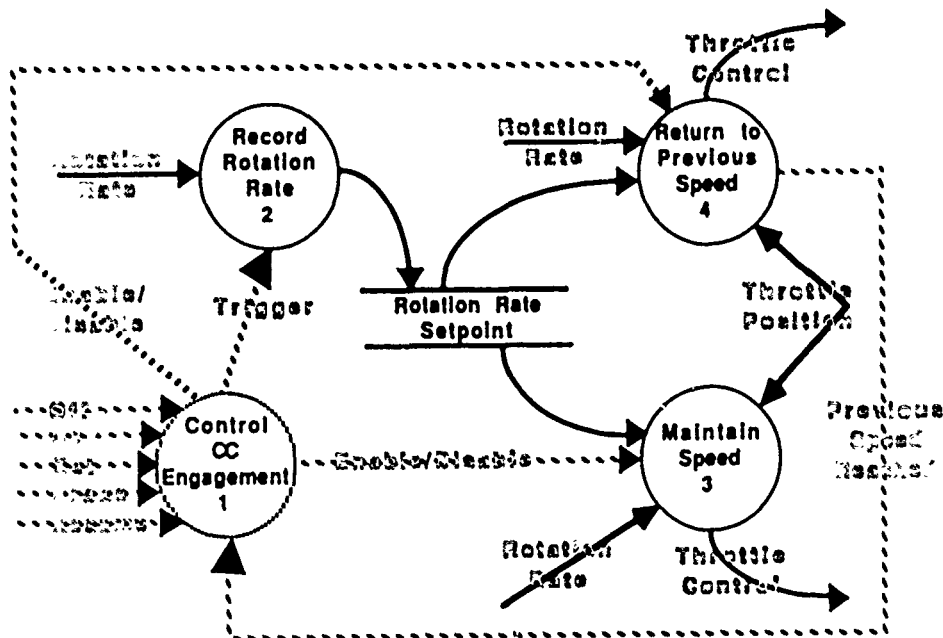
Cruise Control External Entity Graph
Figure 2

Step 2--MMAIM

MMAIM's next step decomposes the software into the major entities of the problem. These entities are software models of physical things and logical concepts in the problem space. The interactions between these entities are identified and the attributes of these entities and their interfaces are recorded. This results in an Entity Interaction Graph, or EIG. The entities on an EIG are initially assumed to be concurrent.

The top-level EIG inherits the double-line arrows from the EEG. For each external entity on the EEG, a reusable entity is created to interface with and model the external entity. For the cruise control, we would create a driver interface entity which would be connected to all of the external arrows connected to the driver external entity. Similarly, we would create a throttle controller interface entity and a drive shaft sensor interface entity.

These "edge" entities should fairly closely model their corresponding external entities to facilitate reuse. For example, another application may use the throttle controller hardware. The software for this application should be able to reuse the throttle controller interface we will



Cruise Control Level 0 Data Flow Diagram
Figure 3

may be represented by more than one arrow on the data flow diagram.

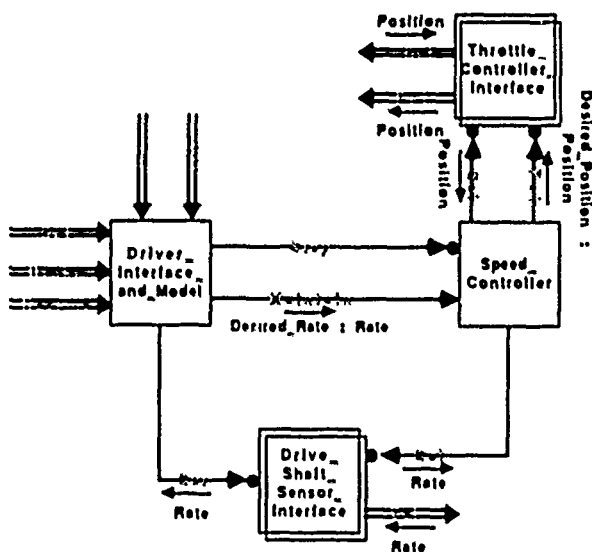
Associated with data flow diagrams is a textual data dictionary. It describes the data flows on the data flow diagrams.

create for the cruise control. Double-line boxes represent reusable entities; single-line boxes represent non-reusable (application-dependent) entities.

The other entities on an EIG model logical concepts in the problem. These are those concepts which the software must handle correctly in order to work. The cruise-control problem has two such concepts. One is the concept of the driver's intentions and the other is the concept of controlling the car's speed. To model these concepts we would create driver model and speed controller entities.

Given the simplifications discussed earlier, we can see that the driver interface entity we created serves no purpose and performs no function. We can obtain the same effect by connecting the external arrows from the driver external entity directly to the driver model entity. I will do this and call the driver model the driver interface and model to reflect this. This change eliminates the reusability concept for the driver interface, but I can't conceive of another use for these driver controls.

Figure 4 shows the resulting EIG for the cruise control. The two reusable edge entities closely model their external entities. The speed controller entity provides two interfaces. One tells the speed controller to start maintaining some desired rotation rate. The other tells it to stop controlling the car's speed.



Cruise_Control Entity Interaction Graph
Figure 4

The speed controller needs to get the current rate so it can compare it to the desired rate. It also needs to command the throttle controller. The driver interface and model needs to determine the rate which the driver intends that the system should maintain. It also needs to command the speed controller.

The remaining feature on this graph is the small black circle at the point of some of the arrows. Called "blocks," these indicate that the logic of the entity to which the arrow points exercises control over when it will respond to a call to the interface. For example, the speed controller will not respond to a call to its stop interface when it is stopped. However, it will always respond to a call to its maintain interface.

At this point a search would be made for existing reusable software components which match the requirements for the reusable entities on the EIG. In many cases these can be found. Applications which obtain the time from a real-time clock may be able to use the predefined package calendar, to name one example. We will assume that none of the reusable entities on Figure 4 exist, and that we will have to develop them.

An EIG provides interface specification information. One of the advantages of Ada is that it provides features to represent interface specifications separately from their implementation. Ada is not perfect in this respect. Ada only represents information about exception propagation and blocks, for example, as comments in a specification. Ada is still a great improvement over languages such as FORTRAN in this respect.

MMAIM provides for mechanical conversion of an EIG to Ada code. By mechanical conversion I mean that the EIG and its associated attribute information provides all of the information provided by the Ada code, and a machine could perform the conversion. In this way MMAIM uses the Ada compiler to check and enforce the software's interfaces from the very beginning of the software-development process. This eliminates many problems with integrating the software, which is a lengthy and expensive phase of most traditional methods.

Before Figure 4 can be mechanically converted to Ada code, we must supply some additional textual information defining the data types on the EIG and the Ada constructs to be used to represent the entities. When this has been done, Figure 4 produces

```
package throttle_controller_interface is
  type position is digits 6 range
    0.0 .. 100.0
;
  function get return position;
  procedure put
    (desired_position : in position)
;
end throttle_controller_interface;

package drive_shaft_sensor_interface is
  type rate is range 0 .. 10_000;
  function get return rate;
end drive_shaft_sensor_interface;
```

```

with drive_shaft_sensor_interface;
procedure Cruise_control is
  task
    driver_interface_and_model
  is
    entry off;
    entry on;
    entry set;
    entry brake;
    entry resume;
  end driver_interface_and_model;
  task speed_controller is
    entry stop;
    entry maintain
      (desired_rate : in
        drive_shaft_sensor_interface.rate
      )
    ;
  end speed_controller;
  task body_driver_interface_and_model
  is separate;
  task body_speed_controller
  is separate;
begin -- cruise_control
  null;
end cruise_control;

```

All the Ada code presented here has been produced by a (human) simulation of a simple program. This results in important aspects of the code, such as meaningful comments, being missing. I have also omitted the Ada constructs which would connect the entries of driver interface and model to their interrupts, since this is hardware dependent.

Now that we have this code, we can do several interesting things with it before continuing the software-development process. By attaching suitable stubs, we can perform top-down testing. Other stubs can produce a prototype system. For large systems, parts of the software can be left as stubs while the rest is completely developed and delivered as a system which provides partial but useful functionality. This partial system may be used while the remainder of the system is developed incrementally.

Step 2--Comparison

These two methods reflect two very different mind sets. The differences between these two mind sets is responsible for the differences between Figures 3 and 4. Finding similarities between them is difficult. Both have the same data coming in and going out of the software. The driver interface and model entity on Figure 4 represents information similar to circles 1 and 2 on Figure 3, as well as the data store. The speed controller entity represents information similar to circles 3 and 4. The other entities on Figure 4 have no counterparts on Figure 3.

Since I assume the reader is more familiar with real-time structured analysis than with MMAIM, I have devoted more of my text to

describing the latter than the former, but it requires about the same amount of time, effort, and cost to develop a good top-level data flow diagram as it does to produce a good top-level EIG. They do require very different mental orientations.

Although they cost about the same to produce, MMAIM includes information not available with real-time structured analysis, and MMAIM provides for mechanical generation of Ada code very early in the development process. With a suitable tool, this code generation could be done automatically. This gives MMAIM the potential to eliminate the coding phase of the lifecycle and to significantly reduce the integration phase.

Step 3--Real Time Structured Analysis

On large problems, some of the data-transforming functions on a high-level data flow diagram will be decomposed into lower-level data flow diagrams. The cruise-control problem is simple enough that Figure 3 is the only data flow diagram. The data-transforming circles on Figure 3 have textual "mini-specifications" associated with them which describe the function of the circle in greater detail.

Control transforming functions on data flow diagrams are not decomposed, but their behavior is represented using a Mealy-model state transition diagram. Figure 5 shows the state transition diagram for the control CC engagement circle from Figure 3.

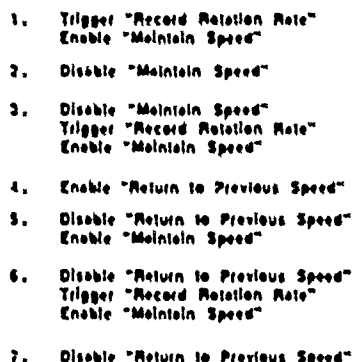
This concludes the application of real-time structured analysis to the cruise-control problem. This has been the traditional analysis phase of software development and takes about twenty percent of the total software-development effort. Real-time structured analysis will be followed by the design and coding phases.

Step 3--MMAIM

On large problems, MMAIM decomposes some of the entities on a high-level EIG into lower-level EIG's. I call these entities "non-primitive." Cruise control is simple enough that Figure 4 is the only EIG.

MMAIM's third step for primitive entities determines the behaviors which the entity performs and the conditions which determine the transitions between these behaviors. Ada context information and declarations required to support the behaviors are specified. This information is given by a Behavior Transition Graph (BTG).

Figure 6 shows the BTG for the driver interface and model entity from Figure 4. Each shape represents a behavior, except for the two top boxes labeled "context" and "declarations." The arrows between the shapes represent permissible transitions between the behaviors. A box represents sequential Ada code. A diamond represents an unconditional wait for an event. A



```

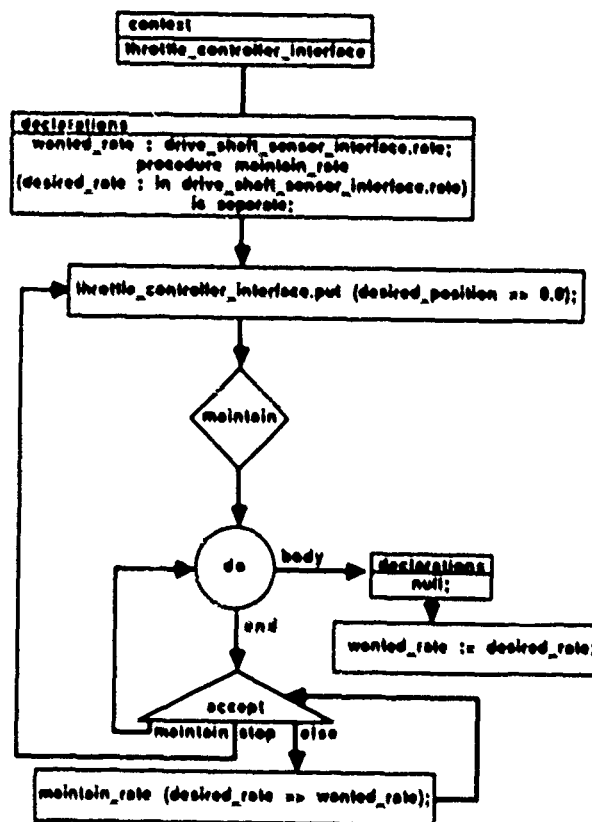
when 2 => -- triangle accept
  select
    accept off;
    MMAIM_behavior := 1;
  or
    accept set;
    MMAIM_behavior := 3;
  end select;
when 3 => -- box
  desired_rate :=
    drive_shaft_sensor_interface.get
;
  MMAIM_behavior := 4;
when 4 => -- box
  speed_controller.maintain
    (desired_rate)
;
  MMAIM_behavior := 5;
when 5 => -- triangle accept
  select
    accept set;
    MMAIM_behavior := 3;
  or
    accept off;
    MMAIM_behavior := 6;
  or
    accept brake;
    MMAIM_behavior := 7;
  end select;
when 6 => -- box
  speed_controller.stop;
  MMAIM_behavior := 1;
when 7 => -- box
  speed_controller.stop;
  MMAIM_behavior := 8;
when 8 => -- triangle accept
  select
    accept off;
    MMAIM_behavior := 1;
  or
    accept set;
    MMAIM_behavior := 3;
  or
    accept resume;
    MMAIM_behavior := 4;
  end select;
end case;
end loop MMAIM forever;
end driver_interface_and_model;

```

Similarly, we can produce the BTG for the speed controller to obtain Figure 7. The speed controller requires visibility of the throttle controller interface, so the latter is named in the context box. The speed controller also uses a variable and a procedure which are declared in the declarations box.

The only new feature on Figure 7 is the circle. It introduces a nested (sub-) BTG. In this case the nested BTG is the critical region which follows the "do" of an accept, indicated by the label "do" in the circle. Nested BTG's do not have a context box, but may have any other feature of a BTG.

Note also the "else" transition from the accept triangle. This corresponds to an else alternative of a selective wait.



Speed_Controller Behavior Transition Graph
Figure 7

MMAIM can mechanically produce Ada code from Figure 7, giving

```

with throttle_controller_interface;
separate (cruise_control)
task body speed_controller is
  type MMAIM_behavior_id is range 1 .. 5;
  MMAIM_behavior : MMAIM_behavior_id :=
    MMAIM_behavior_id'first
;
  wanted_rate :
    drive_shaft_sensor_interface.rate
;
  procedure maintain_rate
    (desired_rate : in
      drive_shaft_sensor_interface.rate
    ) is separate
;
begin -- speed_controller
  MMAIM_forever : loop
    case MMAIM_behavior is
      when 1 => -- box
        throttle_controller_interface.put
          (desired_position => 0.0)
        ;
        MMAIM_behavior := 2;
      when 2 => -- diamond maintain

```

```

accept maintain
(desired_rate : in
 drive_shaft_sensor_interface.
 rate
)
do
  MMAIM_00001 : declare
    type MMAIM_behavior_id is
      range 1 .. 1
    ;
    MMAIM_behavior :
      MMAIM_behavior_id :=
        MMAIM_behavior_id'first
    ;
  begin -- MMAIM_00001
    MMAIM_forever : loop
      case MMAIM_behavior is
        when 1 => -- box
          wanted_rate :=
            desired_rate
          ;
          exit MMAIM_forever;
        end case;
      end loop MMAIM_forever;
    end MMAIM_00001;
  end maintain;
  MMAIM_behavior := 4;
  when 3 => -- circle do
    raise program_error;
  when 4 => -- triangle accept
    select
      accept maintain
        (desired_rate : in
         drive_shaft_sensor_interface.
         rate
        )
      do
        MMAIM_00002 : declare
          type MMAIM_behavior_id is
            range 1 .. 1
          ;
          MMAIM_behavior :
            MMAIM_behavior_id :=
              MMAIM_behavior_id'first
          ;
        begin -- MMAIM_00002
          MMAIM_forever : loop
            case MMAIM_behavior is
              when 1 => -- box
                wanted_rate :=
                  desired_rate
                ;
                exit MMAIM_forever;
              end case;
            end loop MMAIM_forever;
          end MMAIM_00002;
        end maintain;
        MMAIM_behavior := 4;
      or
        accept stop;
        MMAIM_behavior := 1;
      else
        MMAIM_behavior := 5;
      end select;
    when 5 => -- box
      maintain rate
        (desired_rate => wanted_rate)
      ;
    ;
  end

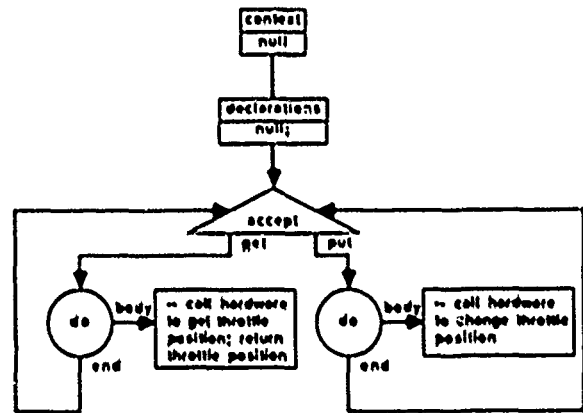
```

```

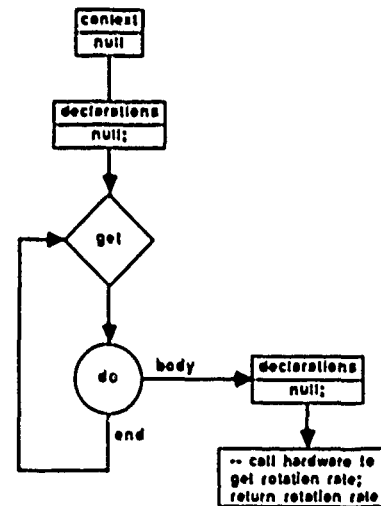
  MMAIM_behavior := 4;
end case;
end loop MMAIM_forever;
end speed_controller;

```

The throttle controller interface and the drive shaft sensor interface are dependent on the hardware with which they interact, so we can't produce real BTG's for them without knowing more about the hardware. We do know enough to outline the structure of their BTG's, which are given in Figures 8 and 9.



Throttle Controller Interface Behavior Transition Graph
Figure 8



Drive Shaft Sensor Interface Behavior Transition Graph
Figure 9

Similarly, we cannot produce Ada code until we have the final BTG's, but Figures 8 and 9 provide enough information that we know the structure of this code. Figure 8 produces something like

```
package body
  throttle_controller_interface
is
  task MMAIM_handler is
    entry get (current_position : out
              position;
    );
    entry put (desired_position : in
              position;
    );
  end MMAIM_handler;

  function get return position is
    current_position : position;
  begin -- get
    MMAIM_handler.get
      (current_position =>
       current_position
      );
    return current_position;
  end get;

  procedure put
    (desired_position : in position)
  is
    -- null;
  begin -- put
    MMAIM_handler.put
      (desired_position =>
       desired_position
      );
  end put;

  task body MMAIM_handler is
    -- null;
  begin -- MMAIM_handler
    forever : loop
      select
        accept get
          (current_position : out
           position;
        )
        do
          -- call hardware to get
          -- throttle position
          end get;
        or
        accept put
          (desired_position : in
           position;
        )
        do
          -- call hardware to change
          -- throttle position
          end put;
        end select;
      end loop forever;
    end MMAIM_handler;
  end throttle_controller_interface;
```

While Figure 9 produces something like

```
package body drive_shaft_sensor_interface
is
  task MMAIM_handler is
    entry get (current_rate : out rate);
  end MMAIM_handler;

  function get return rate is
    current_rate : rate;
  begin -- get
    MMAIM_handler.get
      (current_rate => current_rate)
    ;
    return current_rate;
  end get;

  task body MMAIM_handler is
    -- null;
  begin -- MMAIM_handler
    forever : loop
      accept get
        (current_rate : out rate)
      do
        -- call hardware to get rotation
        -- rate
        end get;
      end loop forever;
    end MMAIM_handler;
  end drive_shaft_sensor_interface;
```

Except for implementing the maintain_rate procedure we declared in the speed controller, this completes the implementation of the cruise-control software. This procedure would get the current rotation rate and throttle position, calculate a new throttle position from them and the desired rotation rate, change the throttle position, and return. The details of the calculation of the new throttle position depend on the relationship between the throttle position and the rotation rate. As this is a function of the hardware, I will not pursue it further.

Step 3--Comparison

MMAIM has created four BTG's compared to one state transition diagram from real-time structured analysis. This is mainly due to MMAIM doing some work at this step which real-time structured analysis defers to the design phase.

There are some similarities between Figures 5 and 6. The "on" diamond of Figure 6 corresponds directly to the "system off" state of Figure 5. Similarly, the top accept triangle corresponds to the "system on" state, and the bottom accept triangle to the "interrupted" state.

There are more differences than similarities between the two, however. The middle accept triangle of Figure 6 corresponds to both of the remaining states of Figure 5. This is because real-time structured analysis' functional orientation separates the very similar and logically related functions of maintaining speed and returning to a previous speed, while MMAIM combines them under the higher-level concept of controlling the speed. This basic difference in

the orientation of the two methods also causes the differences between the actions of Figure 5 and the remaining behaviors of Figure 6.

Another difference at this step, as with the second step, is that MMAIM mechanically produces Ada code from BTG's.

Cost Comparison

Clearly, using MMAIM has required more effort than real-time structured analysis to reach this point. This additional effort produced the three BTG's for which real-time structured analysis has no counterparts. Also, the attribute information associated with an EIG requires more effort from the developer to produce than the data dictionary associated with a data flow diagram. MMAIM appears to require about fifty percent more effort than real-time structured analysis.

MMAIM has completely implemented the software, however, while real-time structured analysis has barely started. Since the analysis phase requires about twenty percent of the total software-development effort, it appears at first that MMAIM only requires thirty percent of the traditional effort to complete implementation, providing a seventy percent savings.

However, this demonstration has not included the effort that must be expended on real projects for testing, documentation, and integration. Although MMAIM effectively eliminates software integration problems, we still frequently see problems integrating the software with the hardware. This reduces MMAIM's savings of effort to about fifty percent. Still, "half price" is a powerful incentive for buyers.

MMAIM has a number of features which contribute to this cost savings. The mind set underlying MMAIM concentrates on such analysis information as major components of the problem and their top-level interfaces, compared to the traditional emphasis on functions and data flows.

Adding top-level design information to the representation of this analysis information eliminates the translation effort traditionally required between analysis and design notations. Coupled with Ada's ability to represent much of this interface specification information as compilable Ada code, this allows coding and testing to begin immediately, as well as providing enforcement of the interfaces.

Finally, the ability to mechanically produce Ada code from MMAIM graphs effectively eliminates the coding phase.

Conclusion

We have seen that Ada, when used with a suitable development method which incorporates the modern software engineering mind set, changes the software development lifecycle. The result is a completely-implemented system without significantly more effort than usually goes into the traditional analysis phase. In addition,

there should be further savings due to reduced maintenance costs, and reductions in future software development costs due to the identification and development of reusable software. These cost savings are an important consequence of the software engineering mind set. It is especially important to note that it appears that not everyone who can produce software in traditional languages, such as FORTRAN, COBOL, Pascal, and C, can develop this mind set [1]. Since software costs are the dominant factor in total systems costs, the use of Ada and the software engineering mind set by appropriate personnel can result in a significant system cost advantage.

References

1. Gerhardt, M., "Don't Blame Ada," *Defense Science and Electronics*, 1987 Aug.
2. Castor, V., and D. Preston, "Programmers Produce More with Ada," *Defense Electronics*, 1987 Jan.
3. Carter, J., "MMAIM: A Software Development Method for Ada, Part I--Description," *Ada Letters*, 1988 May/Jun.
4. Carter, J., "MMAIM: A Software Development Method for Ada, Part II--Example," *Ada Letters*, 1988 Sep/Oct.
5. Ward, P., and S. Mellor, *Structured Development for Real-Time Systems: Volume 2, Essential Modeling Techniques*, Yourdan Press, 1985.



Jeffrey R. Carter
MS L0330
Martin Marietta Astronautics Group
P. O. Box 179
Denver, CO 80201

Mr. Carter is a senior software engineer for the Martin Marietta Astronautics Group. He has been involved with software development for fourteen years and with Ada for four and one-half years. Prior to joining Martin Marietta he developed software in Belgium and England for five years. A member of the Association for Computing Machinery (ACM) and its Special Interest Group on Ada (SIGAda), Mr. Carter was a prize-winner in the 1988 Tri-Ada Hands-On Ada Programming Contest.

THE NATIONAL TRAINING CENTER MOVE AND UPGRADE: A DISTRIBUTED ADA SYSTEM

David L. Pottinger

Science Applications International Corporation
San Diego, Ca. 92121

Abstract

This paper describes a medium sized Ada development activity. The National Training Center (NTC) move and upgrade project involved the move of the Core Instrumentation System (CIS) at Ft. Irwin, California, to a new facility. The project required a redesign of the software and the re-hosting of the system on upgraded hardware. The new architecture was designed to handle near-real-time data rates, digital map graphics, distributed functionality, unit and player symbology, and the recording and display of selected statistical measures.

Introduction

The National Training Center (NTC), located at Fort Irwin, California, is a training facility which provides mechanized and armor battalions an environment for developing basic combat skills. U.S. Army units participate in two week exercises in which they engage an opposition force trained in Warsaw Pact tactics. The exercises are field instrumented and data from the various battles is monitored and recorded for analysis and playback. The incoming data rate is 10K bytes per second.

The NTC Instrumentation System (NTC-IS) provides the means to collect, process, analyze and display performance data. Major NTC-IS functionality includes the control and monitoring of field exercises, recording and replay of field data, control of a defensive live fire range, presentation of statistical reports in support of performance

measures, and the preparation and presentation of After Action Reviews (AAR).

The enhanced system required the use of Ada as the high-level language. The new workstations were to provide a windowing capability and the system was to be composed of off-the-shelf components.

The upgraded system also required the development of nuclear and chemical casualty prediction models for enhanced indirect fire simulation.

Project Description

Current System The current system is about six years old. It is based upon a distributed architecture and is composed of four Vax 11/780s which are quad-ported to four megabytes of shared memory. Two of the machines process and archive incoming data, while two are file servers which respond to requests for data at the workstations. Each workstation is made up of an LSI 11/23 processor, a deAnza graphics monitor and a Hitachi tablet for command input. The software includes about 130,000 source lines of Fortran and 67,000 lines of a special menu description language.

Enhanced System The enhanced system was hosted on Sun Microsystems hardware. There are two compute servers and two file servers. The servers are Sun 3/280s and are attached to two local area nets. Thirty two Sun 3/110 workstations are served by the file servers. All machines use Ethernet protocol for communication.

A commercial database management system was used to load, update and maintain the statistical data.

The new system carried an Ada software requirement. The project was relatively short fused, with a turn-on to delivery period of about 27 months. The software architecture required the development of 50 computer software components (CSCs). At the time of this writing, the project was 30 days away from its initial operating capability (IOC).

Software Development Approach

Considerations Several factors led to the determination of a suitable software development approach. These factors included

- the background of the staff,
- the relatively short development time,
- the existence of a functional baseline system in C and
- the target hardware chosen for the upgrade system.

The staff was primarily composed of experienced C language programmers with a strong background in the techniques of structured design. Several senior staff members were strongly Ada literate. The staff profile was rounded out with entry level personnel having some university Ada experience.

The target hardware system was composed of workstations which were Ethernetted together and which supported the development of decoupled modular processes which communicated with each other on several different functional types of workstations.

The Approach The structured analysis and design techniques as described by DeMarco¹, Yourdon and Constantine² with some modification, were used to develop and document the system design. A commercial automated design tools from Cadre Technologies were used to maintain the design. Ada PDL was used to describe the

detailed design. The Ada PDL was not intended to be restrictive. It provided a means of capturing the design in a form which can be processed by machine, and could be transformed into Ada code. A PDL unique package was developed to support the definition of functions not yet generated, but which had been identified in the design phase.

A prototype system written in C existed prior to the development of the upgraded system in Ada. Several of the CSCs in a baseline system were identified as candidates for translation from C to Ada. A tool developed to convert the C source to Ada. The Ada was then massaged by the original C programmer to perform the same function as in the baseline system.

Appropriate coding standards and conventions were developed and documented. As code was produced, a software quality assurance team insured adherence to the coding standards document.

The Ada development environment was provided by the Verdix Ada Development System. An in-house compiler queuing system was developed which kept the Ada compilers busy, but not overloaded.

A commercial graphics package was selected for use with the workstations. The necessary graphics libraries were developed in the Postscript language. Menu description files, also in Postscript, were prepared by using a proprietary menu layout tool.

Problems Several problems arose which had major impact on the software development process. Of these, the Ada development environment and staff training were the most serious.

Ada training was accomplished through the use of an OJT approach combined with a series of Ada round table discussions with the technical staff. Programmers whose C code was translated into Ada, learned the Ada syntax while massaging the translated source into a form which compiled and executed correctly. Designers not

familiar with Ada, learned the syntax when exposed to Ada PDL. The most Ada literate of the staff were identified as resources to help the inexperienced staff members with problem resolution. The staff was provided with reference books by Booch³, Olsen⁴, and Barnes⁵.

The most significant obstacle to be surmounted was the inadequacy of the Ada development environment. While the resources required for the development of Ada were thought to be understood prior to initial coding, the reality of the situation became clear during the design phase. As more Ada PDL was generated and compiled, the shortage of the secondary storage along with the failing of the compiler to keep up with the demands being made upon it, quickly outweighed the problems with the language itself. The compiler bottle-neck was partially relieved by the purchase of two additional compilers and the development of an in-house Ada queuing system to keep the compilers busy, but not overloaded. The technical staff, which was used to the rapid edit-code-link-execute loop of the C environment, was slow to adjust to the lengthy Ada compiler turnaround time.

Summary and Conclusions

Summary Project results can be summarized with several measures. The design effort produced about 5200 pages of detailed design documentation at a cost of 6250 man days (md). Source lines of code (SLOC) give a feeling for staff productivity and project size. The source lines listed below represent non-comment lines of Ada, C, and Postscript (PS).

SLOC Ada	210,000
SLOC C	16,000
PS library	31,000
PS menu	173,000

Total software development man-days from design through integration is projected to be 17,544md. Not counting the auto-generated menu description code, the Ada, C, and Postscript SLOC were produced at a

rate of approximately 15 lines per programmer day.

The error detection rate gives an idea of the reliability of the NTC software system. At the time of this writing, the error detection rate is approximately one medium or highly critical error per day. A more complete analysis of reliability will be made at the completion of the integration phase.

Conclusions The above project description and results lead to several conclusions about real-world Ada development:

- A significant Ada development project can be successfully designed and implemented by a relatively inexperienced staff.
- the Ada development environment must improve to provide more competition for established development languages like C,
- structured design techniques will produce a workable design for some Ada implementations,
- Ada training can be accomplished without the use of costly and time consuming commercial Ada training programs.

References

1. DeMarco, T. Structured Analysis and System Specification, Prentice-Hall, Inc., 1979.
2. Yourdon, E. and Constantine, L. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall Inc., Englewood, NJ, 1979.
3. Booch, G. Software Engineering With Ada, Benjamin Cummings Publishing Co., Menlo Park, CA, Second Edition 1987.
4. Olsen, E. and Whitehill, S. Ada for Programmers, Reston Publishing Company, Inc., Reston, VA, 1983.
5. Barnes, J.G.P. Programming in Ada, Addison-Wesley Publishers Limited 1984.

6. Ada PROGRAM DESIGN LANGUAGE
(PDL) GUIDELINES, Version 1.1, 20
November 1984 (Document No. SAI
10000-025-0002).

™Ada is a registered trademark of
the U.S. Government (Ada Joint
Program Office)



David L. Pottinger
S.A.I.C.
10260 Campus Point Drive
San Diego, Ca. 92121

Mr. Pottinger has nine years of
Computer Science experience,
including seven years with SAIC. Mr.
Pottinger has managed several
software development efforts on a
variety of hardware suites. The
projects have involved data
processing and management systems,
along with workstation based, C³I,
range monitoring and control, and
tactical training systems.

SOFTWARE QUALITY ASSURANCE IN AN Ada ENVIRONMENT

Shan Barkataki

California State University
Northridge, California

John Kelly

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

Abstract

Use of Ada does not assure software quality. An effective software quality assurance plan remains a necessity for producing quality software. Such a plan for developing Ada software using DOD-2167A is described. Experience shows that this methodology is effective in detecting misuse of Ada as well as finding certain design defects early in the software life cycle.

Introduction

This paper describes our experience with quality assurance work in the design phase of an Ada software development effort using the DOD-2167A methodology [1] at the Jet Propulsion Laboratory at Pasadena. The design methodology itself is described in a separate paper [2].

Ada and Software Quality

Why do we need to worry about software quality when coding with Ada? Doesn't software quality come automatically with Ada? Certainly, Ada has facilities that encourage production of quality software, however, like all tools, Ada can be misused. Use of Ada alone does not assure software quality. An effective quality assurance plan is as necessary as ever for achieving good software quality.

The Inspection Process

The cornerstone of the software quality assurance plan used in this project was a review process that uses the Fagan's inspection method [3]. The inspection process is shown in figure 1.

Defect finding is accomplished in highly structured meetings by a team of

inspectors. In order to help create an ego-less environment, managers do not participate in the inspections. At JPL, the detailed defect descriptions were not regularly reported to the management; they were provided with a statistical summary report at the conclusion of the process.

The Quality Assurance Plan

One major objective of the quality assurance plan was early detection and removal of defects. With this focus, four series of inspections were devised; two during the preliminary design and two for the detailed design phase.

The first inspections were held as soon as the system (CSCI) functions were allocated to the Top-Level Computer Software Components (TLCSCs). These inspections were independent of Ada and concentrated on high-level issues such as completeness of the top-level design, traceability to the requirements and quality related factors such as coupling and cohesion of the TLCSCs.

The second series of inspections were held at the conclusion of the next lower level of decomposition. This is the preliminary design phase, according to the DOD-2167A methodology, when the Computer Software Components (CSC) are produced.

The two major attributes established during the preliminary design were the top-level concurrency (i.e. the number of tasks in each CSC) and the inter-CSC interfaces, including task rendezvous. The design was expressed using compiled package specifications, textual descriptions and entity diagrams giving pictorial representations of the top-level control and data flows. The quality factors evaluated at this stage were uniformity of the design (clarity and maintainability) and general provisions for exception handling (reliability). The package specifications were inspected for

JPL

The Formal Inspections Process

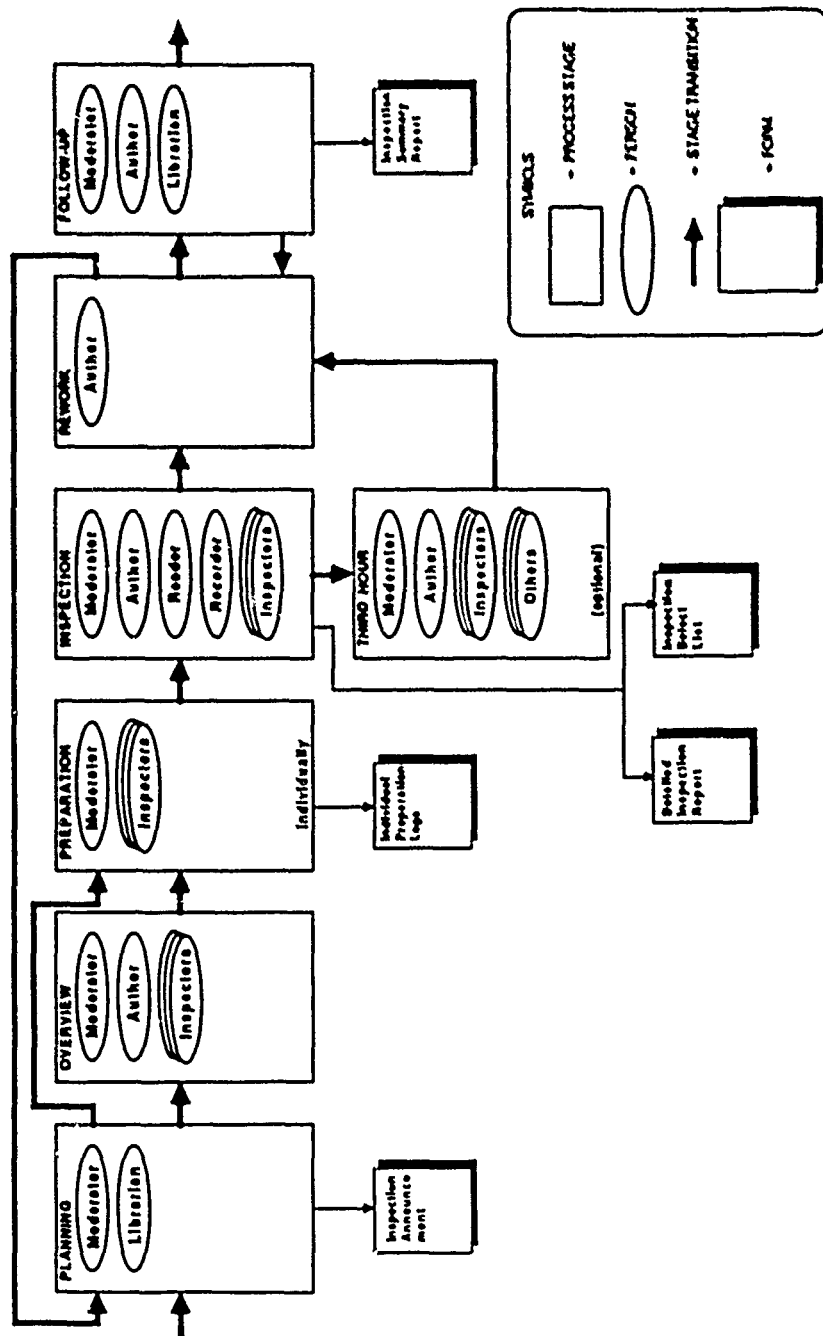


Figure 1: The Formal Inspection Process

information hiding (maintainability) and use of strong types (reliability). Other points of interest were the concurrency and rendezvous provisions. These were inspected from the view point of necessity, completeness and efficiency. The preliminary design review was held at the conclusion of the rework originating from the inspections.

The third series of inspections were held very early in the detailed design, that is immediately after the preliminary decomposition of the CSCs into the Computer Software Units (CSU). The primary focus of the inspections was to examine the design for cohesion and coupling characteristics.

The fourth and final series of inspections were held after production of compiled Ada PDL for the package and subprogram bodies. This marks the conclusion of the detailed design phase. As most design defects had been discovered and fixed previously, the focus during the fourth series of inspections was on quality issues seen at the lower levels, including proper use of Ada facilities. Examples of the Ada related quality factors examined during the inspections were: use of strong types, proper encapsulation of exceptions, information hiding, rendezvous performance, non-use of non-portable features, use of the Ada generic facilities, etc. Clarity and readability of the PDL were also examined. The reworked products were submitted for the DOD-2167A critical design review.

Lessons Learned

The lessons learned are presented in two parts: the first part addresses the issues related to Ada. The second part deals with the use of inspections for quality assurance.

Improving Software Quality With Ada

Our experience at JPL, the California State University and elsewhere indicates that classroom training alone does not result in programmers writing quality Ada code. Currently, many people are first time users of Ada and need guidance on proper use of the language's rich facilities. To some, Ada represents what can be best described as a culture shock .. "Why isn't it enough to produce a program that works"? For these reasons, failure to use some of Ada's powerful features for improving software quality, is a major problem. It is not uncommon to discover Ada code where there is a C or assembler program trying to get out! We found that most problems with non-use

or misuse arise in the areas of exception handling, information hiding, use of strong types, and generic facilities. Use of Ada style guides provide a partial solution to this problem. However, the real solution lies in formal reviews; our experience shows that these can be very effective in improving software quality by weeding out poor or questionable coding practices. A quality assurance plan based on reviews has the additional benefit of providing on-the-job training for software personnel new to Ada.

Improving Software Quality With Inspections

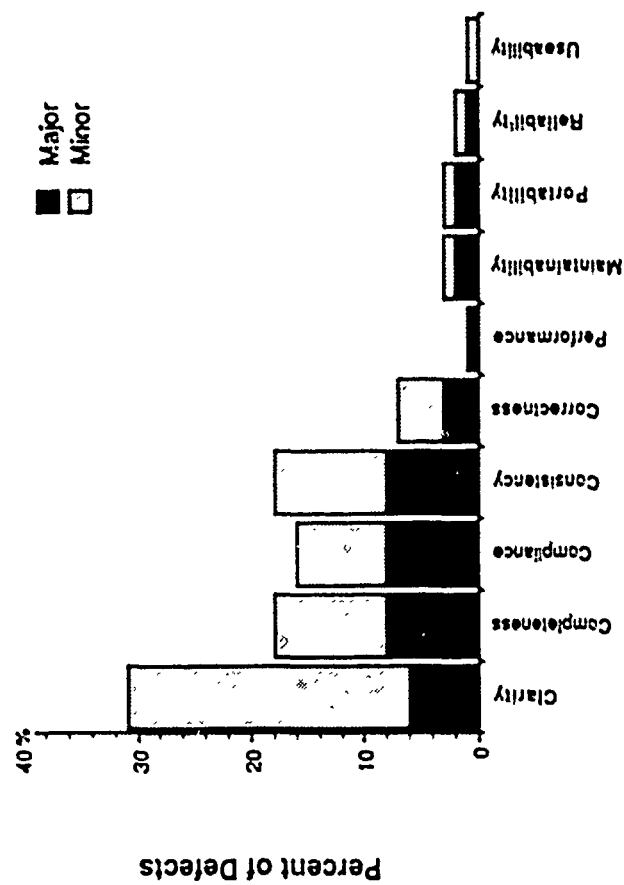
The distribution of defects during preliminary and detailed design phases are shown in figures 2 and 3 respectively. These results were gathered from inspections on a total of 53 different CSUs. Our results indicate that during the design phase, the process is most effective in discovering defects in the areas of clarity, correctness, completeness and internal consistency.

The composition of the inspection team is crucial to its effectiveness. A good team consists of engineers specializing in the various different areas of software development. The inspectors must be "insiders", collectively possessing detailed knowledge of the system being built. The team has to be drawn from people working in the same project but in other areas. The inspection process demands a good deal of the engineer's time. As delivery deadlines approach, inspectors can become a very scarce commodity. It is vital therefore that the manpower requirements for inspections are factored into the project resource plans.

In this project a typical inspection team consisted of the author together with a requirements analyst, tester, user of the software and a software product assurance engineer who acted as the moderator. On average, it took 0.5 work hour per page to complete the inspection process (this includes all time expended by the moderator, reader, recorder, author and other inspectors for all seven phases of the inspections). This translates to approximately 1 person hour to fix each defect. In both cases the time quoted covers all work related to the inspection, i.e. planning, together with finding defects, and fixing and verifying correctness of the solutions.

It is important to establish effective entrance criteria for each inspection. This ensures that the work product being inspected has reached the expected level of maturity and meets the minimum quality

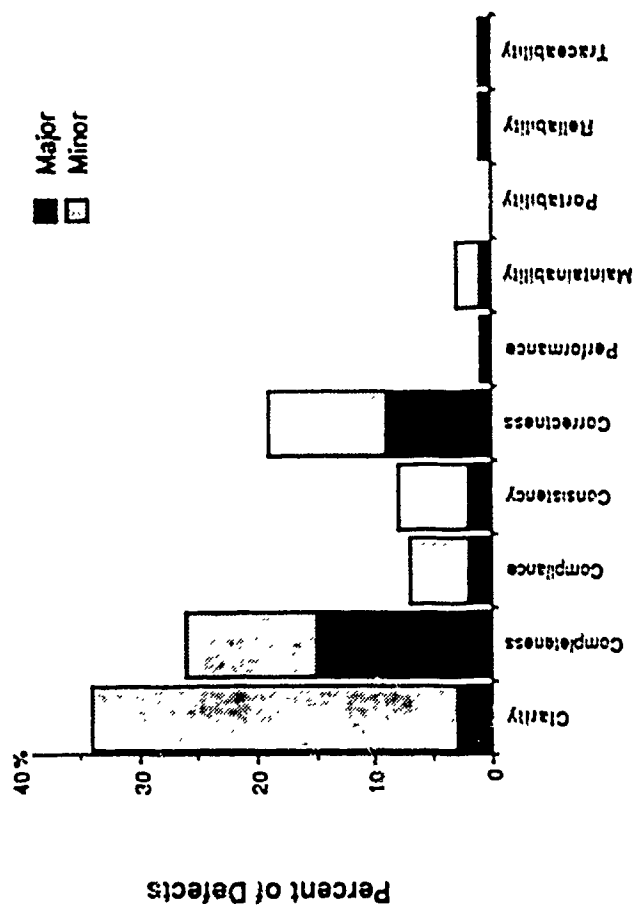
Software Design Document (Preliminary Design) Defect Classification Comparison



Defect Type

Figure 2: Defect Distribution During Preliminary Design

Software Design Document (Detail Design) Defect Classification Comparison



Defect Type

Figure 3: Defect Distribution During Detailed Design

standards. Failure to do this results in poorly focused inspections which consume excessive amounts of time. This is particularly important when inspections are tied to project milestones and when new methodologies and languages are involved.

Conclusions

Our overall experience is that an inspection based quality assurance plan can be very effective in early detection of design and coding defects. Considering that there is almost an order of magnitude escalation in the relative cost to fix errors between the design and testing phases, the policy of early defect removal must pay handsome dividends for the additional investment in effort (4).

References

1. US-DOD; DOD-STD-2167A: Defense Systems Software Development; US Department of Defense, Feb 1988.
2. Ellison, Goulet; " Practical Approach to Methodologies, Ada and DOD-STD-2167A"; Proceedings of the Seventh National Conference on Ada Technology, March, 1989.
3. Fagan M E, "Design and Code Inspections to Reduce Errors in Program Development"; IBM System Journal, Vol 15, no 3, 1976, pp 181-210.
4. Bouha B; "Software Engineering Economics"; Prentice Hall; 1981, pp 39-41.

The work described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration.



Dr. Shan Barkataki

School of Eng and Computer Science
California State University
Northridge, Ca 91330

Dr. Shan Barkataki is a professor of Computer Science specializing in Ada and software engineering. He has worked as a practicing software engineer for over 12 years which has included several Ada projects. Dr Barkataki earned his Ph D in Computer Science in the area of software portability.



Dr. John Kelly

Mail Code 301-475
Jet Propulsion Laboratory
Pasadena, Ca 91109

Dr. John Kelly is a Software Product Assurance Specialist and Chief Moderator at the Jet Propulsion Laboratory. Prior to this he taught Computer Science at various universities. Dr. Kelly has developed and taught classes on the Formal inspection process. His Ph. D. is in Mathematics Education.

IMPLEMENTING SOFTWARE FIRST WITH TODAY'S TECHNOLOGY

Richard J. Gallagher, Jr.
U. S. Army CECOM
Center for Software
Engineering
Ft. Monmouth, NJ 07703

M. Elaine Fedchak
IIT Research Institute
Rt. 26N
Poughkeepsie, NY 12600

David Preston
IIT Research Institute
4600 Forbes Blvd.
Lanham, MD 20706

ABSTRACT

Addressing the software needs of large complex systems first, prior to selection of the hardware components of a system, has been a goal for over a decade and a half. The software first approach requires major revisions to current system development practices. Elements of the software first approach are still beyond our technical grasp. However, the significance of software first is that it provides the baseline into which elements of system development should fit. This paper provides an overview of software first and discusses three elements of this approach with respect to current capabilities. These three elements are requirements definition, uses of prototyping, and development of portable software.

INTRODUCTION

By the early 1970s it had been determined that software had become "the tall pole in the tent." The Air Force budget for fiscal year 1972 indicated that between \$1 billion and \$1.5 billion was spent on software, approximately three times the cost of hardware for the same period [BOEH73]. Not only was software costing more than the hardware component of a system, but it was also believed to be more responsible for schedule slippages, cost overruns, operational penalties and performance penalties, as well as for complex embedded problems that surfaced after system fielding. Both the concept of software first and a software first development machine were proposed in the early 1970s [BOEH73].

Refinements to the theory of software first have been proposed and are currently receiving much attention from programs such as the DoD's STARS (Software Technology for Adaptable, Reliable Systems) program. The commonality of these approaches is the concept of developing the software component of a system, then selecting the hardware to support the system.

The major benefits to developing the software for a system prior to hardware selection include the development of software that will be more maintainable, since all design decisions will be based on what is best for the software; the selection of hardware after the software has been developed, which enables a wiser choice since more is known about the system, as well as permitting the hardware to mature an additional few years while the software is being developed; and the better management of evolutionary requirements since no hardware has been selected and therefore locked into the system.

It should be noted that several of the components of software first do not require a strict adherence to the software first philosophy. Portions of this methodology will, therefore, be usable with a more traditional development approach; other portions will be specifically linked to this methodology. The significance of the software first approach is that it provides a goal toward which various elements of software development should evolve. For example, work on new testing strategies should address the needs of testing portable software while decreasing the sensitivity of the software and testing process to the target machine. By maintaining consistency with a common goal, these evolving strategies will be supportive of a larger methodology and be mutually compatible. Another advantage of this is that much of the benefit of software first can be realized without waiting for all the technology needed to fully implement the model.

To literally implement the recently proposed model of software first [BOEH88] may be beyond the grasp of today's technology in some key areas. Several elements of that model, however, can be supported with existing technology.

The balance of this paper presents an overview of software first, details the use of requirements definition, prototyping, and portable software within the software first concept, and presents conclusions.

Support for this research was provided through RADC contract number F30602-86-C-0111 with funds provided by CECOM Center for Software Engineering.

OVERVIEW OF SOFTWARE FIRST

The traditional method of system development is to choose the hardware for the system, fit the software to the system, then add hardware components, and both add and alter software components to make the system work. By the time the system is fielded, the hardware is several years old and no longer state of the art. The software design has been altered to fit the hardware of choice, often incorporating hardware characteristics into the software design, and has therefore become both machine-dependent and difficult to maintain. And, the schedule has slipped because of this. The negative effects of this approach become even more pronounced during the maintenance phase of the life cycle. In addition, when the system development is completed, often it no longer meets the needs of the user, principally due to the lack of interaction with the user during the development process. As originally conceived, the goal of software first was, and still is, to reduce total life cycle costs. We felt this could be done by increasing user involvement and postponing the selection of the hardware. This seemed easy enough to achieve, in the ideal case, and resulted in the model shown in FIGURE 1. We recognize that software development is not easy and that departures from the ideal model are needed if we are to have a realistic approach. We also realize that the closer to the ideal model we stay, the easier to understand and implement the approach. The result is a plan which allows departures from the ideal model only when necessary and views these departures as ones which should eventually be solved by future technology. In order to reduce implementation time, we propose to use or adopt as much of existing technology (methods, tools) as possible, and to structure the implementation in modular fashion, to allow pieces of it to be implemented and used as soon as possible. Three examples of this modular approach are discussed in depth later in this paper.

As we proceeded with our definition and analysis, we discovered that software first provided solutions to many problems encountered in today's developments and provided the means for implementing ideas which would streamline the overall process further. The end result is the model shown in FIGURE 2. A major benefit of software first is that it forces the system to be viewed as a system as opposed to several systems (hardware, software, logistic) as we do today, and the "systems" to become views of the same system. It also allows requirements and design decisions to be made without unnecessary constraints. To take advantage of this benefit, the software first approach places heavy emphasis on requirements definition.

The purpose of requirements definition is to assure that the user and the developer, and any other key people involved in the development, mutually understand the requirements. Since English is an ambiguous language, requirements written in English are apt to be ambiguous; therefore, it is necessary to iterate the requirements to assure that they are mutually understood. Since complete mutual understanding and agreement is a naive goal, the user and developer must continue to interact during the development to bring to the surface differences in interpretation as early as they are recognized.

Other intents of requirements definition in software first are to ensure that the user is asking for what he needs, that the user and developer understand each other, and that the requirements are feasible with today's technology. An underlying theme of the approach used here is that the requirements focus on what is to be done, not how it is to be done. Only after the requirements have been established are they partitioned into software and hardware requirements. This will facilitate maintaining a focus on the requirements and avoid performing premature system design.

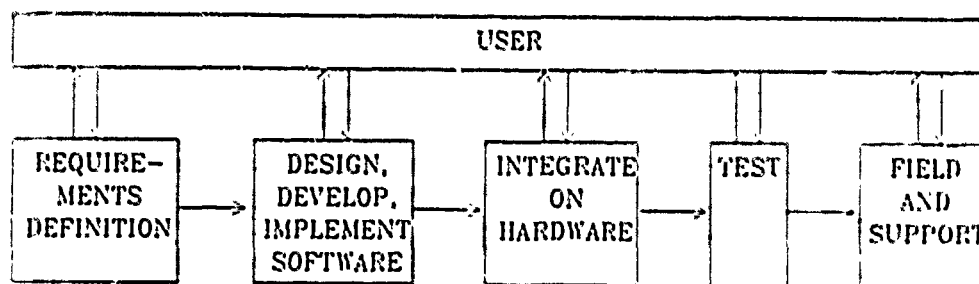


FIGURE 1. THE IDEAL MODEL FOR SOFTWARE FIRST

The Ada language plays a key role in facilitating the requirements definition process, particularly the Ada package. The package makes two significant contributions: first, a package may define a capability that will not be implemented in the initial delivery of the system; and second, a package may ultimately be developed in either hardware or software.

The advantage of putting specific capabilities into packages is that the package is built into the design as part of the initial design process, even if the package is not implemented as part of the initial delivery. Some accommodations must be made, obviously, to ensure that an unimplemented package is not called by the implemented system. Given the nature of the Ada program design languages, the interdependencies of packages are quite easily observed and controlled. The second contribution directly supports the central theme of software first, postponing the selection of hardware as long as possible.

Once system requirements have been developed and are being partitioned into hardware and software, all functions are to be implemented in software unless hardware is clearly superior for implementing the function. The rationale for this is to avoid the necessity of hardware upgrades, and the resulting impact on the system. Changing the hardware which implements specific system functions causes perturbations in the system design; this can possibly be avoided by implementing those functions in software.

In software first, the development host plays a critical role in the success of the project, since all the development, training, and maintenance will be done on it. While this increases the importance of choosing the host, it alleviates many problems which result from the use of multiple hosts (we consider the target a type of host), during the remainder of a system's life. An example is a problem encountered with many complex systems that the intended operator is unable to operate. This occurs when the developed man-machine interface is incomprehensible to the human who is expected to operate the system. The traditional adjustment is to build another layer onto the existing man-machine interface to enhance the system's ease of operation. This additional layer has several negative effects. It degrades performance, it alters the software architecture, it increases cost and delays schedule, and generally aggravates a bad situation. Using the host, the man-machine interface can be implemented and tested using all the capabilities available on the host. Since it is being developed integrally with the rest of the software, implications of changes to the interfaces can be gauged accurately. This allows early visibility into potential problem areas and

allows the user and developer to rationally select the best of the available alternatives, instead of being forced into making a decision under the pressures of a delayed, overrun delivery.

Software first incorporates early training for multiple reasons. First, early training will identify any discrepancies between the skills necessary to operate the system and the abilities of the anticipated operators. Also, by utilizing a trainer, another perspective on the functionality of the system will be obtained which may further identify differences between what the user needs and what the developer is building. Further, with training started early in the life cycle, trained operators will be available when the system is fielded. These changes to the traditional development cycle mandate other changes.

First, in order to begin training early, a prototype of the man-machine interface must exist early. This is required to define exactly the set of functions that the operator of the system must perform and to detail the information presented to the operator by the system. If this early information indicates that the man-machine interface is inconsistent with the capabilities of the operator, then changes can be incorporated into the system.

Training on the host provides other advantages. It provides the opportunity to initiate training before the system has been fielded. This presents the opportunity to have the operators trained prior to system fielding. Also, since maintenance is also performed on the host, the updated system can be introduced to the operators prior to being fielded. This permits the opportunity to determine the effect on the operator of any system upgrades before those upgrades are released. Finally, the host environment is much more robust than the fielded system. Tools that are used for development can also be used to support the training process.

In addition to the training advantages that a prototype of the man-machine interface provides, the prototype enables the user to get yet another view of the developer's perception of the system. Typically, a system's functionality is viewed from the global perspective, which tends to be abstract. This global perspective usually is expressed in terms of the effect of the system on its environment. By defining the man-machine interface, the user and developer can observe exactly how the system will implement the desired effect. This more concrete view of how the system will operate increases the probability that the user and developer truly understand each other.

The system prototype effort will focus on the requirements that are considered "high

risk," and the connection between these selected requirements and any hardware necessary to implement them will be continuously explored. The more general hardware requirements of the evolving software will also be monitored and evaluated. The software first approach requires the software to be developed before the hardware is selected. In practice, it will be necessary to estimate the system's hardware needs early in the development cycle, and assess the feasibility of available hardware to meet these needs. As the software is being developed, the requirements for the hardware which will execute the software will become more clear. As these requirements become clear, the feasibility of existing hardware to satisfy the requirements will be assessed. In this way, the desires of the software world will be tempered by the realities of the hardware world.

In software first, the software is designed, developed and implemented on the host environment. This is to ensure that the software portion of the system functions as intended on the host environment prior to selection of, and retargeting to, the target environment. The software testing takes place on the host environment as part of development, and again later as part of hardware/software integration. Also, all training and maintenance are to be performed on the host environment.

The portability of Ada is a key factor in the integration of the software and the target hardware. The target machine is chosen based upon the needs of the software. While operating on the host, the memory and timing needs of the software are evaluated. Hardware is then selected that can meet these needs. The hardware that is embedded into the system to perform specific system functions is also chosen at this time. By delaying all hardware selections until after the software is developed, the most current hardware can be used in the system.

During acceptance testing the adequacy of the hardware is assessed. If the initial hardware does not satisfy the system requirements, then a change in hardware must be made. The likelihood of the hardware being acceptable, however, is very high as the precise needs of the hardware were identified prior to selection.

The primary significance of the field and support phase is that the support is performed in the host environment. Typically, host environments have access to several sophisticated software development tools. These range from requirements analysis tools, to design tools, to documentation generation tools, to testing tools. These tools are not available in the target environment where maintenance is typically performed. The

benefits of using the host for maintenance is twofold. The tools available on the host make updating the software easier, and the total impact of any changes to the software is easier to assess since the changes are being implemented in the same environment used for development. When a suggested change to system software is made, the impact of that change on the other system software is assessed using the documentation. The change is implemented in the supportive host environment, instead of the austere target environment, and the entire system is retested to assure that changes made have not had an undesirable effect on the system. Of particular interest are potential effects on the system's ability to meet timing requirements.

Several key aspects of software first can be implemented within today's technology. These include enhanced requirements definition, use of prototyping for multiple purposes, and developing portable software. These facets of the methodology will be discussed in detail in the following sections.

REQUIREMENTS DEFINITION

Criteria

There are two assumptions on which software first is based:

Software is flexible and can be altered;

If developed correctly, software can outline several generations of hardware.

Each of these assumptions must be qualified. Although software is flexible, the less it is forced to flex the more likely the software development and maintenance will be successful. To develop software correctly necessitates understanding user requirements and anticipating future requirements. Therefore, to both reduce the need for altering software and to increase the likelihood of long life necessitates doing more thorough requirements analysis.

This section of this paper outlines the criteria for requirements analysis relative to supporting software first; the next section discusses the ability of current tools to satisfy these criteria.

Currently, requirements are typically written in a natural language. These documents vary in length, but are usually long enough to prohibit a detailed understanding of the system requirements by anyone other than the author of the document. This inability to understand requirements documents is due to the ambiguity and lack of specificity of natural language, and the inability to check for consistency or to maintain the document

due to lack of tools. In spite of these shortcomings, requirements continue to be written in natural language due, in part, to the fact that reading natural documents requires no specific training. Because of these shortcomings, there is movement within the software engineering community toward more formalism in requirements documents.

To introduce formalism into requirements documents necessitates a representation scheme. Common representation schemes include finite state machines, program design languages, decision trees, and Petri Nets. The advantages of this introduction of formalism include the virtual elimination of ambiguity and the existence of tools to support the consistency and enhance the maintenance of the requirements document. The disadvantage to the introduction of formalism into requirements documents is that it necessitates training in the specific representation scheme. A discussion on specific techniques for introducing formalism into requirements documents is contained in the next section.

At this time, no single technique for improving requirements documents is dominant. Several exist, each with its own advantages and disadvantages. What is necessary is a set of criteria to evaluate requirements techniques for a particular system development. Three key elements in the evaluation criteria are the user, the developer, and the application.

A set of eight criteria for evaluating a technique have been proposed (DAVIS88):

When the technique is properly used, the resulting document should be helpful and understandable to non-computer-oriented customers and users.

When the technique is properly used, the resulting document should be able to serve effectively as the basis for design and testing.

The technique should provide automated checks for ambiguity, incompleteness, and inconsistency.

The technique should encourage the requirements writer to think and write in terms of external product behavior, not internal product components.

The technique should help organize the document.

The technique should provide a basis for automated prototype generation.

The technique should provide a basis for automated system test generation.

The technique should be suitable to the particular application.

There are additional criteria that must be added to this set to specifically support software first:

The technique should support iterative development of the document.

The technique should support stepwise refinement of the document.

The technique should support the identification of related and interdependent requirements listed in the document.

The technique should support the concept that the man, or operator, is part of the system.

Some elaboration of each of these criteria follows.

The technique must produce a document that is understandable to non-computer-oriented individuals since a high percentage of system users is non-computer-oriented. Many systems produce documents using a formal representation scheme. Since the document must be understood by users who are non-computer-oriented, the representation scheme must be easily understood. The need for some training is inevitable; a very short training session must be adequate or system users will not accept the requirements technique.

Once the requirements document is understood by both user and developer it must drive both the system design and the testing process. The requirements process must focus on what the system is to do and avoid addressing how the system is to do it. This line becomes blurred if, for example, the representation scheme is a program design language. This need to be the basis for a design must not become an excuse for allowing the requirements document to become a design document. Therefore, the system functionality must be evident in the design document without this document constraining the design. The statement that the requirements document should serve as a basis for design must be interpreted to mean that this document exposes the functionality of the system to the design team, not that it defines a particular design. The support for testing comes from the ability to easily interpret the functionality of the system into test cases. Again, the test cases are not to be embedded in the requirements document, but rather the requirements are to be stated in a manner that facilitates the development of test cases.

The primary shortcomings of using natural language for a requirements document are the ambiguity of natural language and the inability to formally determine properties such as completeness and consistency of documents written in natural language. Therefore, the primary capabilities for a technique other than natural language must include checks for ambiguity, incompleteness, and inconsistency. Given the size of the requirements documents of interest, these checks must be automated.

Underscoring the need to focus on requirements and avoid design during requirements analysis, is the criteria that the technique should encourage the requirements writer to think and write in terms of external product behavior. This emphasis focuses the requirements process on what the system will do and avoids the trap of defining how the system will do it. The requirements writer must also consider that the system operator is internal, not external, to the system.

Requirements are frequently generated in a disorganized manner. The individuals involved in determining system requirements may generate a lengthy list of complex requirements. The technique must aid in structuring this list into a coherent document. More specific information on organizing the requirements document is provided with the criteria developed to specifically support software first.

Automated prototype generation and automated test case generation are two capabilities that are gradually becoming available in requirements techniques. These criteria would more accurately be stated as automated support for generation of prototypes and test cases than automated generation per se. The benefit of a formal representative scheme is the potential to automate the translation of requirements stated using the scheme into a prototype or test cases. The need for prototypes during requirements analysis, as stated elsewhere in this report, is to provide additional perspective on the requirements. The need for test case generation is to assure coverage of all requirements by the testing process.

Several of the existing requirements techniques were developed for a specific application domain. Some are applicable to other domains. Part of the evaluation criteria of requirements techniques for a particular system is establishing the suitability of the technique for the specific application. The most direct measure of this suitability would be to establish that the technique had been successfully utilized by the user and the developer previously on a project in the same application domain.

The last four criteria were added to specifically support software first. The software first model proposes a strong commitment to iterative development of the system requirements. This translates into the need of a technique that supports early and frequent changes to the document. Also necessary are version control and archiving of previous versions to enable the reconstruction of a previous iteration of the document if an implemented change is to be deleted. The requirements document in software first will evolve and this evolution must be supported by the technique used to develop the document.

Stepwise refinement is related to iterative development. The stepwise refinement process involves taking a requirement and refining that requirement by adding specificity and detail. This process does not change requirements but rather details the existing requirements. Support in this area includes tracing the evolved requirements, assuring that the refined requirements cover all aspects of the initial requirement, and identifying redundant elements in the set of refined requirements.

The organization and structuring of the requirements document noted earlier has specific implications for software first. Related requirements would include requirements that address a particular high-level function or capacity of the system. Interdependent requirements have correlational or causal links between them. These links must be identified by the technique and exposed, on request, to the user.

Software first supports the concept that the system being developed includes the human with whom the system will be interacting. The implications of this concept include that requirements relative to the user must be included in the requirements document and that the man machine interface is an integral component of the system, not an after-the-fact appendage. Assumptions about the capabilities of the human can be noted easily once the requirements relative to the human are in the requirements document. Automated support for extracting and itemizing these requirements is essential in deducing the capabilities the human is assumed to have. These assumed capabilities can then be compared and contrasted to the anticipated user requirements. Discrepancies can then be addressed as appropriate.

Tools

Requirements analysis has been singled out as a key process in software first, worthy of new and better techniques for defining requirements. Therefore, existing tools for requirements analysis have been studied for applicability.

Automated tools for requirements analysis may be categorized in a number of different ways. Some tools have been designed to automate the generation and maintenance of what was originally a manual method and these tools typically make use of a graphical notation for analysis. This class of tools produces diagrams, aids in problem partitioning, maintains a hierarchy of information about the system, and applies heuristics to uncover problems with the specification. More importantly, such tools enable the analyst to update information and track the connections between new and existing representations of the system. For example, a number of CASE (Computer Aided Software Engineering) tools enable the analyst to generate data flow diagrams and a data dictionary and maintain these in a database that can be analyzed for correctness, consistency, and completeness. In fact, the true benefit of this, and of most automated requirements tools, is in the "intelligent processing" that the tool applies to the problem specification.

Another class of automated requirements analysis tools makes use of a special notation (in most cases this is a requirements specification language) that is processed in an automated manner. Requirements are described with a specification language that combines keyword indicators with a natural language narrative. The specification language is fed to a processor that produces a requirements specification and, more importantly, a set of diagnostic reports about the consistency and organization of the specification.

After the system-level requirements have been defined through user-developer interaction and the use of requirements analysis tools, the next step in the software first approach is to partition the requirements into software and hardware requirements.

Thanks to recent strides in microprocessor technology and in software engineering (e.g., the development of Ada), the size and complexity of embedded real-time systems have grown explosively. It is now practical to implement many functions in software that earlier would have been relegated to hardware. This flexibility has led to the software first bias toward software implementation of a function whenever possible. When the project begins, the designers do not know the functional system division between the hardware and the software. Tools that supply the hierarchical functional decomposition framework can be used to help define exactly which functions should be performed in hardware and which in software.

Several of the tools that implement the common methods for requirements definition and requirements analysis have recently been evaluated against the criteria defined in the preceding section [DAVI88]: output that is understandable to non-computer-oriented users; output that forms a basis for design and testing; automated checks for ambiguity, incompleteness, and inconsistency; system view is in terms of external behavior; output should be organized; tool should support automated generation of prototypes and test cases; and the tool should support the specific application. The tools are based on either finite state machines, decision tables or decision trees, program design language, structured analysis, or Petri nets.

Finite state machines appear to provide the superior representation against the stated criteria, with the only major drawback being the amount of training to understand the tools and their inputs and outputs. Decision tables or decision trees are most appropriate for decision-intensive applications, and tools based on this model do not provide for automated checking of requirements for ambiguity, incompleteness, or inconsistency. Further, these tools do not provide support for prototype or test case generation. The strength of program design language tools is that they are intuitive, due to their similarity to natural language; the weakness is the lack of formalism necessary to assure well structured documents to drive well structured designs. Static analysis can be performed to detect structural errors, but this is more beneficial in the design phases than in requirements definition. Structured analysis tools, even those extended to support real-time applications, are more appropriate for systems based on data flow and data structure. This leads to a data-flow view of the system, not an external view. Static structural and behavioral analysis can be performed, but the structure is data driven. Petri nets are strong at representing synchronous behavior, but appear to be hard to master, particularly for the non-computer-oriented user.

Tools based on the finite state machine model appear to be the most supportive of requirements definition process detailed in the previous section.

PROTOTYPING

The proposed approach to software first relies on prototyping to accomplish several goals. These include requirements definition, communication between user and developer, definition of the man-machine interface, early availability of training devices, and determining feasibilities of parts of the

system. This section discusses several aspects of prototyping relative to software first.

Motivation

Requirements are difficult to state explicitly and completely at the outset of a project. The user may know the need but be unclear about the details of the solution. Users and developers must deal with inherent communication boundaries; what one means to say may not be what the other hears. Waiting until later in the project to detect and correct a major misunderstanding is expensive in terms of money and time. The concept of prototyping encompasses a variety of specific techniques which facilitate communication and understanding between users and developers.

The man-machine interfaces of a system are particularly critical to the success of its development effort. They are complex, and contain a wealth of detail. Prototyping is beneficial for determining detailed input and output requirements and man-machine interaction requirements. The use of prototypes can help identify and discriminate among which functions the user can effectively control and which functions the system needs to perform automatically. Prototypes of the man-machine interface can also be used for early training.

Several characteristics of system developments influence the relative benefits of using prototyping techniques:

- o Application area
- o Application complexity

o Customer characteristics

o Project characteristics

"In general, any application that creates dynamic visual displays, interacts heavily with a man user, or demands algorithms or combinatorial processing that must be developed in an evolutionary fashion is a candidate for prototyping" (PRES87). Many COCOM systems display such characteristics, and are therefore candidates for prototyping even if they are not following a software first approach. Prototyping is, however, critical to software first.

When building a system using new concepts or technologies Brooks (BRO075) contends

"... even the best planning is not so omniscient as to get it right the first time. The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers..."

Pressman contends that a prototype can serve as the throwaway system (PRES87). This moves prototyping from a luxury if one has the time to an expected and required learning expense that can be recognized and minimized. The proposed software first system development approach echoes this conclusion.

The remainder of this section describes several kinds of prototyping, and relates them to the model of software first presented in FIGURE 2.

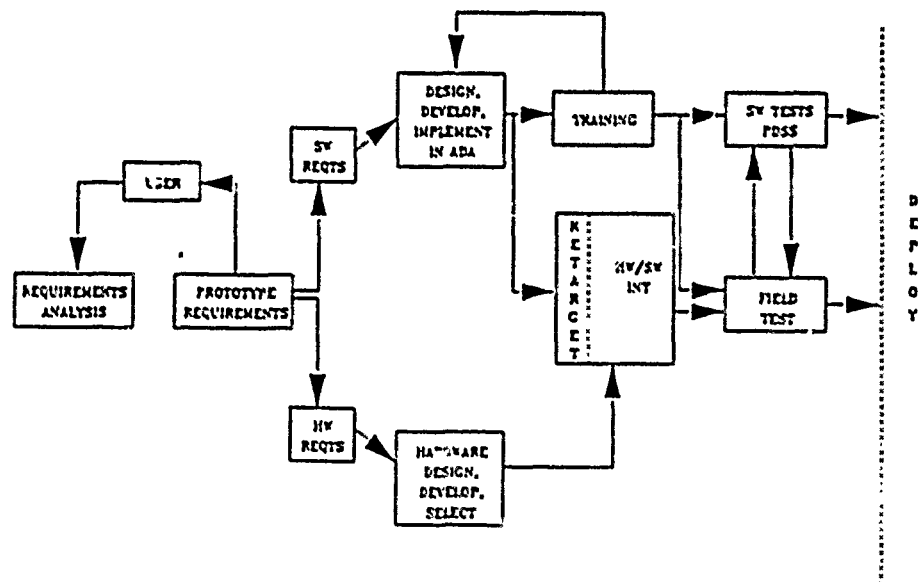


FIGURE 2. THE PROPOSED SOFTWARE FIRST SYSTEM DEVELOPMENT MODEL

Paper Prototypes

Paper prototypes are images on paper of what the terminal screens, reports, physical interfaces, and actual system will look like. Paper prototypes provide a useful and inexpensive way to improve communication during requirements definition. They are appropriate from the earliest stages of system definition. They allow the interface requirements to be determined before any analysis or design is done. They result in solid requirements for the man-machine interfaces and in a reduced chance of misunderstanding-induced requirements changes later in the life cycle. Another consequence of using paper prototypes is that documentation of the man-machine interfaces, inputs, and outputs will be developed very early in the life cycle rather than after the system is built.

The software first approach to requirements definition stresses a need for the developer and user to understand each other and to agree on the system definition. The paper prototype approach fosters this interaction by requiring the developer to spend extra time with the user explaining the paper prototypes and listening to the user's comments.

Nonfunctional Prototypes

A nonfunctional prototype is software that accepts all valid inputs and displays a sample of each of the outputs, but has no functionality behind the displays. For instance, the software may accept a digitized image from an external source and display it on the screen. When the user requests an FFT and filtering of the image, however, another prestored image is displayed because the processing software to do the FFT and filtering functions has not yet been implemented.

Within a software first approach, nonfunctional prototypes provide the same advantages as do paper prototypes: improved communication and better requirements definition. The eventual users of a system can use nonfunctional prototypes to get a feel for the system being developed. If an appropriate man-machine interface is not obvious from discussion or paper prototypes, the developer can provide a few different nonfunctional prototypes which employ different man-machine interfaces. Feedback from the users can then help determine the proper approach. More importantly, the users can determine whether or not the man-machine interface maps to their conceptual model of the task to be accomplished.

Training early in the life cycle with user feedback to the developers is a key

component of the proposed software first approach. This can be accomplished by using nonfunctional prototypes to provide the initial training for system users.

Techniques for building nonfunctional prototypes include the use of prototyping languages, 4th generation languages or conventional languages. The code developed for nonfunctional prototypes may be reused in the system, although the primary purpose of nonfunctional prototypes is to help define the system requirements, not to develop an initial implementation of the system.

Working Prototypes

Working prototypes have interface capability and limited functionality. Working prototypes are usually disposable. They are usually built with a 4th generation language or by exploratory programming. They are used to explore feasibility issues, such as whether or not a time constraint can be met.

Working prototypes are constructed from the high level design. Their applicability to the software first approach is in questions of feasibility. They may also be used to test design options. Working prototypes are generally meant to be used as a learning tool and then discarded.

Development Prototypes

A development prototype is a program that provides part of the desired functionality. The functionality is then augmented until the system is completed. This approach is also known as incremental development with user involvement. The software design must be available for the development prototype to be built.

Development prototypes may be used to establish requirements and then discarded. More commonly, however, they are used to provide a continuous prototype for the user, as the prototype is developed into the complete system. Such a continuously available, evolving prototype can provide a mechanism for user interaction throughout the development cycle, as emphasized in the software first approach.

Prototyping has also been used to test various approaches to solving a problem. This technique is also known as exploratory programming. A developer who is unsure as to how best to design or implement a new technology or deal with a new requirement may turn to prototyping to acquire some experience in the area before committing to a particular development path. For software first, this applies to exploring high risk requirements, especially those that affect the eventual selection of hardware for the system.

Prototyping Tools

The goal of prototyping is to learn the most about a problem for the least cost. This translates into wanting results without spending a lot of time coding. This issue has been addressed by recent developments in techniques for rapid prototyping. Rapid prototyping encourages the use of very high level languages and the reuse of code. Included in these categories are 4th generation languages, prototyping languages, demonstration languages, and object oriented development environments such as Smalltalk and Lisp environments. The software first system development approach requires the use of such tools to achieve its goals of improved requirements definition, better communication between users and developers, more careful consideration of the man-machine interface, and early training, through the use of prototyping.

PORTABILITY

Portability is a key ingredient of software first. It is both a driver of the philosophy and a goal of the development approach. The development of Ada as the first portable programming language, along with the recognition that software lifetimes should not be prematurely shortened by tying them to hardware lifetimes, together make software first both possible and appealing. The desire to have application software outlive its initial target hardware, or developing, training, calibrating and maintaining the software on the host environment, transform software portability from a nice ideal into a requirement.

This section discusses portability definitions, implications, suggestions for increasing it, and finally measuring it.

Definition

Software portability is the ease with which correctly functioning software running in an environment can be made to correctly function in another environment. An environment in this context means the hardware and operating system that the application software runs on.

Although software is often classified as portable or nonportable, in reality portability is a measure of the middle ground between two extremes. The smallest amount of effort to do a port is to recompile the software for the new target environment. The other extreme is to translate or rewrite the entire application while still retaining the original design. If the software application needs to be redesigned and rewritten it is considered nonportable. Nonportable software must be redeveloped.

Benefits of Portability

Portability provides flexibility. It allows the customer to make use of preferred hardware and operating systems. It also allows the upgrade of hardware and software as dictated by changing needs. An environment upgrade could, for example, result in the ability to process the same amount of data faster.

Computer hardware technology has and is expected to continue to improve at a rapid rate. This has several effects:

- higher clock rates
- faster memory
- new architectures (RISC, TRACE)
- new devices (optical disks, OCRs, military unique devices)

In this climate, the environment chosen as the target at a project's inception may be obsolete by the time the system is fielded. Portability provides the flexibility to adopt ongoing technology advances. This flexibility allows for greater choice (and competition) in selecting hardware platforms.

Portability provides cost savings, particularly in the maintenance phase of the software life cycle. Software with high portability will have few if any changes when a new release of an operating system or when a hardware upgrade is adopted.

A new class of hardware may become available with improved price performance. Porting the software to this new hardware will allow the system's performance to grow with hardware advances without incurring complete redevelopment costs. New software technology in compilers, development and maintenance tools, and operating systems will benefit portable systems. Taking advantage of hardware and software technology advances extends systems' useful lives. All these factors contribute to cost savings.

Portability saves time. The DOD is often looking for ways to reduce the length of the development cycle. Porting existing systems to new hardware and incrementally extending the systems capability is usually quicker than building a new system from scratch. This revitalization approach obviates some development efforts and provides an interim capability until new systems are fielded. In this way, portability can save time and complement the development cycle.

Challenges

Portable software is not created by accident. To deliver highly portable software the developer must consciously address the following:

language portability
software developers' knowledge of machine dependencies
documentation of environment dependencies and assumptions
modular and parameterized dependencies

Language compilers must be available for the target machines. The language semantics need to be unambiguously defined. The language implementation must not require uncommon hardware features. For portable applications it is beneficial to use a language which has a standard, such as Ada. Vendor unique language extensions decrease portability.

Programmers need software development experience to generate portable code. Programmers typically go through three levels of competence. First, they write the code so that it does what they want it to do. Second, they understand how it is that the code that they write instructs the machine to perform appropriately. Third, they understand the differences among target environments and write code that will do what is desired as independent of the environment as possible.

For example, seemingly innocuous statements can present portability problems:

```
if (A(x) and B(x)) then ... endif;
```

This statement has several interpretations:

If A(x) is FALSE then B(x) will not be executed
Either A(x) or B(x) may be executed first

If the first is FALSE the other will not be executed.
A(x) and B(x) will always be executed

In some applications the answers to these questions will effect the results and/or the performance of the application. Experience is important in writing portable code.

Dependencies and assumptions need to be documented. Documenting serves two purposes: to make the developer more aware of the dependencies in the code and to aid developers in porting the code in the future.

Assuming that environment dependencies will occur in the code, there still are steps that can be taken to keep portability high. One approach that works well is to localize the nonportable features in modules separate from the rest of the functional code. When the software is being ported, attention can be focused on the few modules that contain the dependencies rather than scanning and changing the code throughout the system. Encapsulation of hardware interfaces into modules also has the same benefit.

Another approach to increasing portability is to parameterize rather than hard code information. This is an accepted software engineering principle that directly affects portability. An example is where there is a location in memory that has information that the system accesses from many places in the code. Directly (hard) coding the address at each place it is referenced works, but portability would be increased if that address were defined as a constant in one place and the constant used everywhere else.

Software portability is affected by the nature of the function performed by the software. Techniques to improve performance are often directly the opposite of portability recommendations. Embedded systems often have unique hardware interfaces for the devices with which they work.

Another trade off is developer "productivity" versus portability. If there is a software deadline, developers will spend less effort on portability in order to spend additional time to develop functionality. This is because most customers give functionality higher priority than portability.

Approaches to Improve Portability

Software that verifies inputs and results is easier to port. Robust coding aids in determining when assumptions are false in a new target environment. This can save debugging time when doing a port.

Use available standards when feasible. De facto standards are also helpful. Use a well known approach rather than an equally effective home grown approach.

The code should be easily understandable and maintainable by a less experienced person. Understandable code is valuable when the person who does the porting is different from the person who did the development.

Use an approach which will work on the broadest class of machines. The "standard" may differ between two environments. In order to be portable a different subroutine may have to be called to perform the same function.

The design should be built such that it avoids depending on environment unique features. The design needs to modularize the environment dependencies. The designer should consider whether the unique facilities can be emulated if the software is ported to another environment.

Performance Versus Portability

Experienced developers will trade off some efficiency for portability considerations, because they know that it is the cost effective approach in the climate of ever faster/larger/cheaper hardware.

Portability can be improved by using only those language constructs and subroutine calls which are supported by all environments (all the environments that the system is intended to run on). This is known as common intersection or minimal support level. This means that the software has to run on the poorest environment or even a subset of the poorest environment.

For mundane programming, software can be made very portable. For embedded systems and/or time critical systems high portability is difficult or impossible. "When performance is of greatest concern, and a large number of calculations are required, it may be beneficial to use range constraints that translate exactly to common underlying hardware (16 or 32 bit) to allow the compiler to utilize hardware overflow detection during operations" (GRIDS88). The current approach with embedded systems is to get the most capability and performance from the hardware available. The developers will apply every trick possible to provide the most capable system with the hardware they have. The requirements for the system should specify where on this portability/performance spectrum the customer would like the system to be.

Long term solutions to the portability/performance tradeoff are continually being sought. One approach is to have military standard computers rather than a different architecture for each system. Another approach is for the compiler to deal with generating the most efficient code. Compilers continue to improve but in most cases are still not as good as a human doing this. Ada is especially susceptible to this since Ada compilers are only about 4 years old.

FIGURE 3 depicts the hypothesized portability/performance curve. The line is theoretically the best that can be done. Systems to the left of the curve should be avoided because they can obtain additional portability without sacrificing performance.

In reality this graph may have many parallel curves or contour lines, where each line represents how much money the customer is willing to spend. Additional funding can buy some improvement and shift the curve to the right.

Real-time performance is an area where portability difficulties are obvious. Hardware speeds vary and what runs 5 seconds on a Cray 2 will take longer on an IBM PC. If the application is insensitive to or flexible on the time it takes to run then it is more portable. Timing quirks also occur, differences in device speeds can cause problems with servicing and missing interrupts or being interrupted at unexpected/inconvenient times. This can happen if the new target runs slower or faster than the original environment.

Measuring Portability

Portability is a function of the software, the original environment, and the target environment. It is sometimes expressed as the amount of effort required to be able to port the software. This 'amount of effort' metric is influenced by the knowledge, experience, and capability of the people performing the port.

The 'amount of effort' metric can be used with the analogy approach. The first port of the system can be estimated using porting

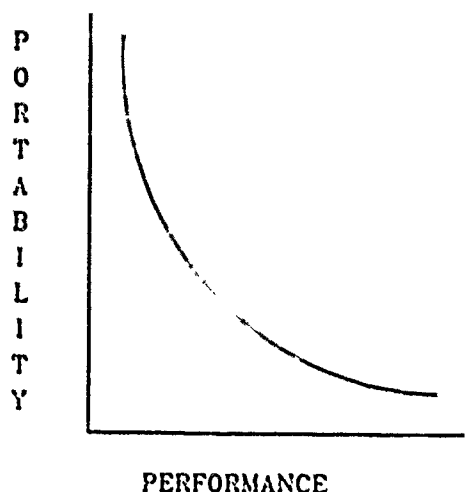


FIGURE 3. PORTABILITY VERSUS PERFORMANCE

costs for similar systems and personnel. Later ports can be based on similar systems and the port history of this system. However, this approach cannot be used by the customer in evaluating the portability of the software delivered.

Several reports are available which address specific techniques to use and other techniques to avoid when writing portable Ada code [1815A], [GRIE88], [MATT87], [NISC82], [FAIT85].

CONCLUSIONS

One primary objective to stating the full methodology for software first is to provide a framework into which individual elements of the software development process can fit. In this way, the elements, although perhaps developed independently, will be consistent with a single approach to system development and, therefore, be mutually compatible. Also, by using individual elements of the software first development approach, some of the benefits of software first will be achieved.

Several elements proposed by software first can be implemented without adhering to the software first approach. For example, prototyping the man machine interface will be beneficial irrespective of the development approach. Software first necessitates the development of a man machine prototype to clarify requirements, to support early training, and to determine if operators will, in fact, be able to operate the developed system.

Enhancing the requirements definition process is another element essential to software first but valuable regardless of the development method. A shift toward formalism in requirements definition is necessitated by software first to decrease the ambiguity of requirements, to enable use of tools for consistency checking, and to enhance the maintainability of the requirements document. These benefits are being realized by developers today.

Prototyping is accepted almost universally within the software community as a valuable tool with multiple applications. For software first these applications include avoiding and resolving differences between user and developer over interpretations of requirements; establishing feasibility of high-risk requirements; and prototyping the man machine interface. The man machine interface prototype will be utilized and will therefore need to be a robust, functional prototype. Prototyping is essential to software first but, obviously, can be utilized without adherence to the software first approach.

Ada is the most portable language to have been developed due to the standardization process for Ada compilers. In part due to Ada, portable software is becoming a reality. Measures for portability exist and are maturing. Guidelines for enhancing the portability of software exist; most focus on isolating the implementation-dependent portions of the system into a limited number of modules. One major benefit of this enhanced portability, irrespective of software first, is porting software to a common support environment for post deployment support.

REFERENCES

- [1815A] Ada Programming Language, ANSI-MIL-STD-1815A, Department of Defense, January 1983.
- [AGRE86] W.W. Agresti, "SEL Ada Experiment: Status and Design Experiences", Proceedings of the 11th Annual Software Engineering Workshop, NASA/GSFC, December 1986.
- [BOAR84] B. Boar, Application Prototyping, Wiley-Interscience, 1984.
- [BOEH73] Boehm, Barry. "Software and Its Impact: A Quantitative Assessment," *Datamation*, Vol. 19, No. 5, May 1973.
- [BOEH87] Boehm, Barry. "Improving Software Productivity," *Computer*, Vol. 20, No. 9, September 1987.
- [BOOC83] Booch, Grady. *Software Engineering with Ada*, Benjamin Cummings Publishing Company, Menlo Park, CA, 1983.
- [BOOC87] Booch, Grady. *Software Components with Ada*, Benjamin Cummings Publishing Company, Menlo Park, CA, 1987.
- [BRIC87] D. Bricklin, DEMO II, (software), Software Garden, 1987.
- [BROO75] F. P. Brooks Jr., *The Mythical Man-Month*, Addison-Wesley, 1975.
- [BROO87] Brooks, Frederick. "No Silver Bullet - Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4, April 1987.
- [BROO88] Brooks, Craig, Edward J. Gallagher, Jr., and David Preston. "The Software First System Development Methodology," Proceedings of the 6th National Conference on Ada Technology, Crystal City, VA, March 1988.
- [CAST87] Castor, Virginia and David Preston. "Programmers Produce More With Ada," *Defense Electronics*, June 1987.

- [DAVI88] Davis, Alan M. A Comparison of Techniques for the Specification of External System Behavior," *Communications of the ACM*, Vol. 31, No. 9, September 1988.
- [DEDA86] De Bartolo, Gil and Ron Richards. "The Back-End of a Multi-Target Compiler," *Proceedings of the 4th National Conference on Ada Technology*, March 19-20, 1986.
- [FALK88] Falk, H. "CASE Tools Emerge to Handle Real-Time Systems," *Computer Design*, January 1, 1988.
- [FELS88] R. C. Felsing, Object Oriented Design, seminar notes, 1988.
- [FLAI76] Flaisher, Robert J. "Software-First System Design," *Compocon76*, February 24-26, 1976.
- [FRED83] Freeman, Peter and Anthony Wasserman. *Software Design Techniques*, IEEE Computer Society Press, Los Angeles, CA, 1983.
- [GOLU82] Golubjatnikov, Ole. "Architecture, Hardware and Software Issues in Fielding the Next Generation DoD Processors," *Proceedings of the 2nd AFSC Standardization Conference*, December 1982.
- [GRIE88] L.J. Griest and T.E. Griest, Preliminary Transportability Guideline for Ada Real-time Software, LANTEK Corporation, Woodbridge, CT 06525, April 30, 1988.
- [HOLL86] Hollan, J.D., E.L. Hutchins, T.P. McCandless, M. Rosenstein, and L. Weitzman, Graphical Interfaces for Simulation, Institute for Cognitive Science Report 8603, University of California, May, 1986.
- [IITR86] IIT Research Institute, "Establish and Evaluate Ada Runtime Features of Interest for Real Time Systems - Interim Report", Contract MDA 903-87-D-0056, U.S. Army CECOM, March 1988.
- [KELC84] P. Kruchten, E. Schonberg, and J. Schwartz, "Software Prototyping Using the SETL Programming Language", *Software*, Vol. 1, No. 5, October 1984.
- [MACD82] MacDonald, Alan, "Visual Programming," *Datamation*, Vol. 28, No. 11, October, 1982.
- [MAHA87] Mahajan, L., M. Ginsberg, R. Pirschner, and R. Guilfoyle, "Software Methodology Catalog," Technical Report MCD7-COMM-ADP-0036, U.S. Army Communications-Electronics Command, October, 1987.
- [MANT88] M.M. Mantel and T.J. Teorey, "Cost/Benefit for Incorporating Human Factors in the Software Lifecycle", *CACM*, Vol. 31, no. 4, 1988.
- [MATI87] E.R. Matthews, "Observations on the Portability of Ada I/O", *ACM Ada Letters*, Vol. VII, no.5, 1987.
- [MCF88] G. McFarland, P. Brennan, J.D. Litke, M.S. Restivo, "A Tool Set for Distributed Ada Programming", *Grumman Data Systems*, Woodbury, NY, 1988.
- [MCGA88] F.E. McGarry, W.W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)", *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988.
- [MONA78] Moralee, Dennis. "MIL-SPEC Computers - Building the Hardware to Fit the Software," *Electronics and Power*, August 1978.
- [MORI84] Moriconi, Mark, Representation and Refinement of Visual Specifications in PEGASYS, RADC-TR-84-129, June 1984.
- [NASA87] National Aeronautics and Space Administration, Goddard Space Flight Center, "Ada Runtime Environment Issues", preliminary report, June 1987.
- [NIEL88] K.W. Nielsen and K. Shumate, "Designing Large Real Time Systems with Ada", *CACM*, Vol. 30, no. 8, 1987.
- [NISS82] Nissan, Wallis, Wichmann, and others, "Ada-Europe Guidelines for the Portability of Ada Programs", *ACM Ada Letters*, Vol. 1, no.3, 1982.
- [PAPP85] F. PAPPAS, Ada Portability Guidelines, SofTech Inc., Waltham, MA, March 1985, DTIC/NIS #AD-A160 390.
- [PRES87] R.S. Pressman, *Software Engineering-A Practitioner's Approach*, second edition, New York, NY., McGraw-Hill, Inc., 1987.
- [POON88] Poonen, G., H.M. Nachiappan, S.O. Landstrom, "A Graphic Design Assistant for Ada and Information Systems," *Proceedings, 6th National Conference on Ada Technology*, 1988.
- [SCH85] J. Schill, R. Sreaton, R. Jackman, "The Conversion of Commands & Control Software to Ada: Experiences and Lessons Learned", *Ada Letters*, Vol. IV, Issue 4, 1985.
- [SIVL87] Sivley, Karen E. "Experience and Lessons Learned in Transporting Ada Software," *Proceedings of the Joint Ada Conference Fifth National Conference on Ada Technology and Washington Ada Symposium*, March 16-19, 1987.

[TAF788] Taft, Darryl K., "Lab Supporting Development of Ada Tasking Tools", Government Computer News, April 15, 1988.

[TATE85] G. Tate and T. Docker, "A Rapid Prototyping System Based on Data Flow Principles", ACM SIGSOFT Software Engineering Notes, Vol. 10, no. 2, April 1985.

[WIE84] R. Wiener and R. Sincovec, Software Engineering with Modula-2 and Ada, New York, NY., John Wiley & Sons, Inc., 1984.

[WILL87] Williams, T. "Real-Time Development Tools Aid Embedded Control System Design," Computer Design, October 1, 1987.

[WOOD86] Woodside, C.M., "The CAEDE Performance Estimator for Structural Designs of Ada Programs," 9th Minnowbrook Workshop on Software Performance Evaluation, August, 1986.

BIOGRAPHIES



Mr. Edward J. Gallagher, Jr. is currently the Chief of Systems Software Technology Division, Advanced Software Technology, CECOM Center for Software Engineering, where he is responsible for research and development efforts covering software reuse and Ada applied to real-time systems and the associated area of runtime environments. In his previous assignment with Project Manager PIRS/TIDS he was responsible for all the software for the PIRS/TIDS hybrid, a complex position location/communication system. He has also served as the chairman of the STARS Human Resources Area Coordinating Team and has written and reviewed Ada and computer resources policy for the command. He received his B. S. in Electrical Engineering from Carnegie Mellon University, and an M. S. in Management Science from Fairleigh Dickinson.



Ms. M. Elaine Fedchak is a Research Computer Scientist with IIT Research Institute, where she is on the professional staff of the Data & Analysis Center for Software. Her current research interests are in the areas of software engineering methodologies, tools and standards. Her recent work includes development of a test standard and a background investigation for the development of a software engineering environment in which she analyzed the information requirements of DDO-STD-2167 and its DIDs. Ms. Fedchak received her B. S. degree in Mathematics from St. Lawrence University in 1979.



Dr. David Preston is a Senior Software Engineer with IIT Research Institute and an adjunct faculty member of the Computer Science Department of the University of Maryland. His specific research interests are the use of Ada for secure systems, real-time application issues, and runtime environment evaluation techniques and criteria. He is a member of IEEE, the IEEE Computer Society, and ACM. He holds a B. S. in Earth and Space Science from Clarion State College, an M. S. in Mathematics from Ohio University, and a Ph. D. in Mathematics Education and an M. S. in Computer Science from the University of Maryland.

LESSONS LEARNED IN DEVELOPING REQUIREMENTS

Garlan Healer

Lockheed Engineering and Sciences Company
Space Station Freedom-SSE System Project

Abstract

This paper describes lessons learned from developing a requirements specification document for a large complex Ada system. The requirements specification process is too often glossed over by the engineers so they can dive directly into the design of the system. A properly completed requirements specification will drive the entire life-cycle of the system. The purpose of this paper is to describe the development architecture, methodology and methods used during the requirements specification process of the Space Station Freedom Software Support Environment (SSE) Project. The SSE requirements specification process is explained with its problems and with some suggestions for a CASE tool to handle the requirements analysis phase.

Requirements Specification Life Cycle

The requirements specification phase is the beginning of the life-cycle which effects the entire life-cycle of the system. The purpose of the requirements specification is to contain a complete description of the systems functions without describing how the system is implemented, serve as the basis for design activities, and the basis for system test planning.¹ Figure 1, Software Engineering Life-Cycle, shows how the software requirements phase in the software engineering life-cycle should be implemented. The software requirements specification affects the entire system life-cycle because the design and testing of the system are developed from the requirements specification. The software requirements specification affects the entire life-cycle and a mistake in this area can mean costly mistakes in the other phases of the life-cycle.

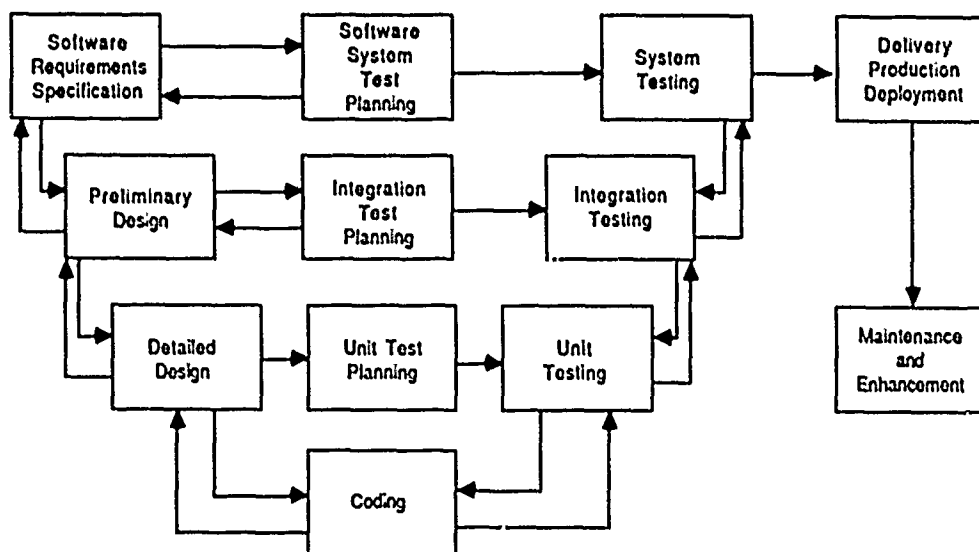


Figure 1 - Software Engineering Life-Cycle

What is requirements specification?

The requirements specification is a method used to describe what functions the system will perform, a means for describing the workings of a system, and a way for the customer to understand the purpose of the system. The requirements specification is also used as the system basis for the design and testing. According to Yadav, Bravoco, Chatfield, and Rajkumar⁵ the requirements specification should be described in the form of: a functional model of the object system; a data dictionary defining the various components of the functional model; and set of performance and test specifications for the system.

What takes place in the requirements specification phase is the requirements analysis process and requirements specification documentation production. The requirements specification phase of the life-cycle needs to adopt a methodology for requirements analysis, and a process to produce the requirements specification.

What is requirements analysis?

The requirements analysis is the part of the requirements specification phase that deals with analysis of the requirements to ensure the systems objectives are met through the requirements, and that there are not any unnecessary, unwanted or ambiguous requirements. There are three basic questions with respect to requirement analysis: What should the requirements be (content); How should requirements be stated (content form); How should the requirements be derived (content determining process).⁵

Several structured techniques have been developed to help an analyst model at the requirement determination level; Structured Analysis and Design Technique, Data Flow Diagram (DFD), Business Information Analysis Technique, Integrated Definition Method, Interpretive Structured Modeling Software. It is not clear to educators and practicing professionals which techniques are better suited for requirements analysis.⁵ These techniques provide testing for conflicting requirements, ambiguous specifications, incomplete requirements, redundant requirements, protocol-deadlocks between subsystems, and various other redundancies in the system

specification. Using an object-oriented approach for the system design can be awkward if structured analysis techniques were used when defining the requirements, since the criteria for grouping functions are different. The transition from one to the other may require significant recasting of the DFD's. This is a laborious process, which can be avoided by assuming an object-oriented viewpoint during the analysis phase.⁶

What are the requirement specification document contents?

After the requirements analysis phase of the life-cycle is complete a method of reporting and writing the processes from that phase are essential to the requirements life-cycle and system development phase. The requirements specification document addresses the findings from the requirements analysis. A way to represent and describe the requirements analysis is the purpose for the requirements specification document.

The requirements specification phase of the life cycle will produce several requirements specification documents. The requirements specification document identifies the purpose of the system and provides an operational scenario of how the system will be used.⁴ The requirements specification document contains information for the system users, designers, implementers, and testers and it should include the following information.⁵

- Functional specification of what functions the system must perform,
- System context, Constraints, and Assumptions,
- Performance specification about the dynamic properties of the system,
- Measurement and test conditions for a organized testing process to verify that the system is behaving properly.

The requirements specification document should include context analysis, functional specification, and design constraints. Also different types of requirements should be addressed in this document for system design and testing.

The next questions are: What methodology best suits the requirements specification phase and how to produce the requirements

specification document containing the appropriate information.

Introduction to the SSE Project

The SSE System consists of host and workstation computer hardware, systems software, communications networks, and SSE software. These components, when integrated and operated as a networked system, will provide the total life-cycle management of the Space Station Freedom Program software. The SSE consists of software, standards, hardware specifications, methods, procedures, documentation, and training capabilities.

The contract for the SSE project provides the Space Station Freedom Program with a software environment to aid in producing flight and ground software for the space station. The SSE provides the tools, rules, and procedures along with an integrated software environment to support the development of application software.

The SSE project is task to deliver various types of documents throughout the projects life-cycle. The SSE Preliminary Requirements and SSE Detailed Requirements specification documents are the primary SSE requirements documents being delivered to the National Aeronautical & Space Administration (NASA). Knowing that requirements will change and may change often these documents will be updated annually, after the baseline is approved by NASA.

The SSE contract requires the production of approximately 90 documents during the initial two year term of the project. This fact brought about a great need for a controlled document development and production process to handle the many developers working on various documents.

The SSE System set up in house for the document development process consists of three different types of workstations, Apollo Series 3000, Macintosh II, and IBM PS/2 Model 60. These workstations are connected to a DEC VAX 8820 using an ethernet Local Area Network (LAN). The LAN also includes the connection of the laser printers which produce hard copies from the workstations.

The workstations are configured with word processors, graphics software, and CASE tools for documentation development. The IBM PS/2 and MAC II are configured with the Microsoft Word word processor and the Apollo with Interleaf as its word processing tool for document development. The graphics software used is GEMDraw for the IBM PS/2, MacDraw for the MAC II, and Interleaf for the Apollo. The CASE tools available are PowerTools on the MAC II, Teamwork on the Apollo, and Execelerator on the IBM PS/2.

The LAN uses the File Transfer Protocol (FTP) for transferring developed files from the Apollo and the MAC II to the VAX, and uses Kermit for files transferring from the IBM PS/2. The files are then transferred and stored from different systems and applications using interoperability filters. Information on tool interoperability filters is located in RISC Symposium'88 proceedings under "Tool Interoperability in SSE OI 2.0".⁷

On the VAX the documents are controlled using the Automated Product Control Environment (APCE) which was developed and supported by the subcontractor Planning Research Corporation (PRC). The APCE provides an environment for storing, testing and configuration management functions for the documents.

This process works because the developers use the workstations for word processing, DFD, and data dictionary development of their assigned sections of the documents. Then the sections in a file are transferred to the VAX where the developer loads the sections into the APCE. Once in the APCE, the section is ready for testing. A tester checks the format and content of the developed sections using the APCE as a test tool. If something is incorrect, the developer is informed of the changes that need to be made through VAX electronic mail. This process continues until the entire document has passed the testing process including final integration tests. To produce a hard copy of the document, a VAX host-based tool called SCRIBE is used to print the document on a laser printer.

SSE Methodology in Developing Requirements Specification

The methodology used for developing a requirements specification document for systems using Ada begins with careful preparation of the annotated outline. A group of three and four develop the annotated outline using a boiler plate similar to the DID's (Data Item Description) described in DOD-STD-2167 for addressing requirements. The annotated outline identifies major sections, appendices, system users, and reference documents. The systems requirements sections in the annotated outline briefly describe what requirements will be included in that section. The sections in the annotated outline are analogous to a process of the system.

A book manager is in charge of assigning the appropriate people for developing the different processes of the system and is in charge of bringing the document together. The book manager delegates each section to a group who will develop the requirements. A context diagram of the system is provided to develop DFDs for the section. The context diagram is usually obtained through the concept document, proposal, or according to customer specifications.

A kick-off meeting is conducted with the people (developer & testers) involved in the development of the requirements specification document. During the kick-off meeting the book manager informs the developers of their assignments and passes out standard formats and procedures defined to guide the developers during the requirements specification development process.

The developers begin collecting relevant requirements on the section from system users, contractors, reference documents, and the customer. All the relevant requirements are collected and merged into a file. The developers analyze collected requirements to identify processes which are sections within the document. The DFDs are developed for processes and subprocesses which are analogous to sections and subsections of the document. Active entities should appear as processes, passive entities as data flows; every function must be performed by some entity.⁶ The introduction of each section of the document contains the mini-spec for the process it is associated with in the DFD. The mini-spec

describes the actions performed on the data flowing into and out of the processes. The developers decompose requirements, processes, and DFDs iteratively to accomplish functional partitioning, functional decomposition and testing of the system.

The requirements associated with each process are grouped under that section. The Functional Requirements explain what function the element will provide, the Detail Requirement clarifies the functional requirement, the Single, Detailed, and Testable Requirements specify how fast, how many, and how frequently the functional requirement will perform. Figure 3, Requirements Specification Document Structure, shows the lower level document sections and the types of the requirements which correspond to the sections.

In the initial development phase of the requirements, a complete, consistent and accurate statement of requirements for a system may be impossible. The reasons are: inability of users to foresee all levels of detail, complexity of the system; inconsistency between various user viewpoints.⁴ For the unknown areas TBD's (To Be Determined) will be substituted until additional data for the areas can be defined.

The methodology is iterative in nature so that the final requirements specification document includes all primitive subprocesses and DFDs for each subprocess. This methodology allows for an easy transition to the system design process.

Once the document is initially completed the customer and user community are allowed to review the requirements through a Review Item Description (RID). The RID is a form that allows the customer to voice concerns with requirements within the document. The RIDs must then be incorporated into the document and again reviewed by the customer. When the customer approves the document, the document is then considered baselined.

In practicing the above methodology, the developers and testers encountered several problems. The problems and recommendations are presented in the next section as lessons learned in developing requirements specification documents for a large Ada system.

Problems Developing Requirements Specification

The methodology explained above, though fairly rigorous, was not without problems. This section discusses the problems that were encountered during the requirements specification production process and provides several recommendations to avoid future problems. These problems are described below.

- During the specification process frequent changes occurred to the annotated outline while document development was in progress. This was caused by a lack of management understanding of the requirements specification and production process. Predefined standards for requirements specification documents similar to DOD-STD-2167 should be used produce an annotated outline. A CASE tool that accommodates a frequently changing outline is required. Documentation bread board and streamlining methodology would also help.
- Developers' personal expertise was not always utilized by the book managers assignments. Making a list of the expertise you need and match the staff expertise to the section assignments would make this process easier.
- Book manager of the document needs to organize informal meetings and forums for participation by the customers, and user community with the developers of the requirements specification document.
- Lack of training in the supported methodology caused inconsistency problems within the document. Training for the developers in the chosen methodology, and in analysis techniques is required before starting development of the requirements specification process. Enforce project standards for DFDs and provide a CASE tool to adhere to these standards and methods.
- DFDs and data dictionaries were inconsistent due to lack of understanding of analysis techniques among the requirements specification document developers. The use of a master data dictionary for developers would prevent duplicate naming conventions and definitions.
- Global terms were inconsistently defined in the document. Create a global glossary for defining the terms used in the requirements specification document, also the glossary needs to be controlled and maintained by the book manager independent of the requirements specification developers.
- Decomposing of requirements and DFDs to a testable level for starting the design process based on Ada. Not all of these requirements will be available during the initial requirements specification phase. Inform the customer of the missing areas and proceed as directed.
- The traceability to each primitive requirement to the reference document requirements was not well supported by the environment, "traceability and reporting using manual methods is a painful process".³ According to Stanton³ development managers, project leaders, and engineers will spend over 40% of their time preparing compliance documents, and most of that time will be spent tracing requirements. A CASE tool that automates the recording of traceability for requirements that change during the development, and after baselining of the requirements specification document, is necessary.
- Reorganization of the document in the supporting environment was difficult. Make sure your CASE tool supports user friendly document reorganization and that the traceability pointers move with the requirements.
- Deadlines were negotiated by management and NASA at the start of the contract award date, not by the developers. This decision hurt the consistency, accuracy, usefulness, and completeness of the document. However often the requirements development phase is constrained by a deadline and a solution gets picked without taking the time to research what the customer requires.⁴ It seems like "there is never enough money to do it right the first time but there is enough to do it right latter". It is best to plan on more than one iteration of the document before baselining the requirements.

- Requirements of the system are dynamically being changed by the customer during the requirements specification process. Also note that it is not uncommon for the customer (often the Government) to want to change "The Requirement" during development.⁴ As new versions of the parent documents are released updates to the existing version of the requirements specification need to be provided through a CASE tool.

Many of the problems could be solved using a requirements management tool. Actually the biggest problem is establishing the requirements management capability at the beginning of the program and making requirements management part of the whole life cycle.² A comprehensive CASE tool for requirements management to support the methodology is necessary although presently there are only tools to support documents and DFD's independently.

Requirements Analysis CASE Tool Criteria

The manual processes in the requirements analysis phase can be reduced with a CASE tool that avoids having to manually scan, interpret, and input documents, create and maintain links and pointers, control configuration management procedures, and transfer files to and from other systems.² A CASE tool that automates the manual procedures can mean significant savings in the amount of time spent in the requirements specification phase.

A CASE tool that integrates the capabilities of word processing, structured analysis, and object-oriented design, that allows for automated generation of the requirements specification document which includes traceability for the requirements, and DFD generation corresponding to the sections in the document. A CASE tool with those qualifications is what was required during the requirements specification phase on this project.

A CASE tool for requirements analysis does not just provide links and pointers, that is only the beginning. To fully support requirements management, the environment must also have integrated text and graphics editors, configuration management and version control facilities, boilerplate requirements tracing forms,

enforce standards and control deliverables, a central design database with multi-user access and control.²

The following criteria is suggested for a CASE tool to be used during the requirements specification process.

Case Tool Criteria

- Provides configuration management and version control facilities.
- Provide for automatic checking in the following areas: Static Structural-signals transmitted through inappropriate ports, or signals received by an entity but not sent by any entity, two requirements conflict; a specification is ambiguous, incomplete, or redundant, and protocol-deadlocks between subsystems.¹ Natural language does not provide for automatic checking so a Requirements Specification Language must be used.
- The ability to trace requirements throughout the system development life cycle would help solve many of the significant problems facing developers and managers such as: Planning, Development, Verification, Communication, Project Control, Change Control, and Documentation.²
- Create a requirements history report and be able to report the status of all requirements in compliance reports.²
- Integrated text and graphics editors within the CASE tool
- Integrates structure analysis and object-oriented design techniques within the tool.
- Allow creation of boilerplate requirements tracing forms.
- Has a central design database with multi-user access and control.²
- Interactive entry and viewing of traceability pointers for each individual requirement.
- Individual requirements within different versions of a document and across different versions of configurations.²
- Integrated Automated document production capability.
- Allow reconfiguration of the environment to enforce the adopted methodology and standards of the project.
- Interface between a requirements management system and a document management system or a document generation system.²

- Automatically generate system level tests directly from the requirements.¹
- Integrate a planning tool into an environment for planning of the life-cycle phases

Conclusion

Requirements management is a must to streamline the development and maintenance of a large system. In requirements management we are talking about system design analysis and being able to assess the impact of an engineering change request (CR).

The problems pointed out above were not deficiencies in the methodology, but rather the need for planning, organization, developer training, and the use of automated methods with a CASE tool to impose constraints and standardization during the development process. The solutions described above can be provided with the use of current methodologies and with the use of a CASE tool which integrates word processing, structure analysis, object-oriented design and is able to trace requirements throughout the system development life-cycle.

Acknowledgements

I would like to thank Amerash Sripathi for his help in collecting information and for co-writing the summary to this article. Also I would like to thank Pat Schill for helping me make this article into a professional looking product.

References

1. Alan M. Davis, "A Comparison of Techniques for the Specification of External System Behavior", Communications of the ACM vol 31,9 (Sept 1988)
2. Dave Sharon, "Requirements Management for Building Better Systems", Nastec Corp., Vol II, No. 3 (Summer 1988)
3. Tom Stanton, "Requirements Management and the RTRACE Environment", Nastec Corp., Vol II, No. 3 (Summer 1988)
4. George E. Sumrall, "Requirements Engineering and Ada", Proceedings of the Sixth National Conference on Ada Technology, (March 1988)
5. S. Yadav, R. Bravoco, A. Chatfield, & T. Rajkumar, "Comparison of Analysis Techniques for Information Requirement Determination", Communications of the ACM vol 31, 9 (Sept. 1988)
6. Sidney C. Bailin, "An Object-Oriented Specification Method for Ada", Proceedings of the Fifth Washington Ada Symposium, (June 1988)
7. C. L. Carmody & C.T. Shotton, "Tool Interoperability in SSE OI 2.0", RICIS Symposium '88 Houston, TX (Nov. 1988)
8. DOD-STD-2167, "Defense System Software Development", June 4, 1985

Garian G. Healer
Lockheed Engineering
and Sciences Co.
1150 Gemini A22
Houston, TX 77058
B.S. Computer Science
from Texas A&M
University in 1985

Work for Lockheed on NASA's Space Station Freedom Software Support Environment (SSE) project in the modeling and simulation area. Have been involved in Ada work for the space station. Also have worked on military applications for Lockheed Missiles and Space Co. Current interests are requirements specification.

TANGRAM₁ - A PROGRAM DESCRIPTION LANGUAGE FOR ADA

Ernst-Erich Doberkat

University of Essen, Department of Mathematics
Computer Science/Software Engineering
Schützenbahn 70, D-4300 Essen / West Germany

ABSTRACT

We describe the program description language Tangram₁ in some detail, in particular we show how Tangram₁ is integrated into an object oriented approach to software design in Ada. Tangram₁ is based essentially on a blend of SETL's very high level diction and Ada's data and program abstraction facilities. We indicate how Tangram₁ may be used for program generation as well as for reusability.

Introduction

Programs can become quite complex, and it helps to have a sound methodological approach to their construction. In the case of Ada* there seem to be two competing approaches for managing this complexity: the more traditional functional approach, and the object oriented one.

The functional approach is an extension of the well known top down design method in which functions and functionalities are the primary objects of consideration. Booch argues that this approach is not adequate to the linguistic capabilities of Ada, and suggests the object oriented approach [Boo, Ch. 2]. The latter approach is based on objects and their behavior as well as on program abstraction "which provides operations on an object whose representation and identity is hidden from the user". [Weg, p. 257]. Wegner points out that the functional approach - and correspondingly the functional programming style - lends itself to support working with functions in the mathematical sense (i.e. working without side effects). The object oriented approach on the other hand might be compared with using mathematical machines (like finite state automata) in which not only the input but some internal state which is hidden from the outside affects the output. From a practical point of view, Booch suggests the following steps in an object oriented development:

1. Identify the objects and their attributes.
2. Identify the operations suffered by and required of each object.
3. Establish the visibility of each object in relation to other objects.
4. Establish the interface of each object.
5. Implement each object.

* Ada is a registered trademark of the U.S. Government (Ada Joint program office)

(1802, p. 171). Step 3 establishes the outside view of each object, and step 4 its inside view, so that for the purpose of this discussion both steps may be merged into a single step *Establish the views for each object*. In constructing the views, the connection with step 2 is crucial - it has to be firmly established what the operations on each object are in order to implement them. The operations associated with each object should be described in some formal manner before they are implemented. This formal description may serve as an *abstract blueprint* (as a blueprint, so to speak), but under favorable circumstances may be used, too, as a *basis for the derivation* of the implementation itself. If the description is of a sufficient high and formal level, and the transformations can be shown to be correctness preserving, it may even be used to *formally verify* the implementation. Since a description is supposed to contain the algorithmic content of a package, it may be used, too, for *supporting the reader* of the package's code.

The practicality of such a formalism for describing Ada packages hinges on convenient means of expressing the contents of such packages (i.e. the objects implemented, and the operations on and for these objects). The description should be *concise* (otherwise much of its effect is lost), *mathematically oriented* (thus giving a basis for verification and transformation), and *supportive of program abstraction* (in this way supporting the object oriented paradigm).

In this report we propose a program description language (called *Tangram*) blending the very high level diction of SETL with the data and program abstraction facilities of Ada. Thus we advocate using finite set theory as an adequate way of describing the functionality of the operations under consideration. The SETL experience has shown that set theory is a very adequate notational tool for describing the functionality of an algorithm without cluttering the description with implementation details (see e.g. [Kru], [SDDS]). This has the obvious advantage for the programmer/designer that he may concentrate on algorithmic - rather than implementation - details. A SETL specification is executable, but SETL's advantages of notational convenience have to be paid for by a usually very poor performance. This disadvantage is obviated by a combination of tools to transform SETL programs

- the RPTS-system transforming high level SETL to low level SETL (see [Pai]),
- the translation of SETL to Ada, converting executable specifications to production efficient programs (see [DoGu]).

SETL's concepts of data types in comparison to Ada's makes it difficult, though, to blueprint Ada programs (or packages) directly in SETL. Thus for capitalizing on the descriptive power of set theory in the context of object oriented design with Ada, it is desirable to combine it with Ada's data abstraction facilities.

These remarks make it evident that *Tangram* uses an approach which is rather different from the one used in designing ANNA (see [LKHO]): ANNA is a language extension to Ada; it works by annotating Ada constructs. In this sense, ANNA is quite close to Ada, since important Ada concepts (scope and visibility, elaboration, generic instantiation) may be applied to ANNA's annotations. We will see that this is not the case with *Tangram*. Both ANNA and Asphodel (see [HIL]) are languages with the goal of supporting the design of Ada programs. Asphodel seems to be based on VDM and may be used in an annotational way, too. Its main emphasis seems to be on the formal verification of specifications, and it uses models of objects, which seem to be similar to abstract data types. It remains to be seen what differences and similarities there are between *Tangram* and Asphodel.

This paper is organized as follows: Section 1 deals with the intent of the proposed language. Here the dual purpose of blueprinting programs and supporting reusability are discussed, and we briefly digress by discussing some of SETL's constructs. Section 2 contains the language description proper. Here we discuss in some detail the language's descriptive mechanisms in the context of the overall organization of a *Tangram* module. We finally offer a brief sketch of the *Tangram* system as a system planned to support reusability.

Acknowledgement: Most of the work described here was done while the author was on the faculty of the University of Hildesheim. The author wants to thank Ms. S. Karmrodt for her skillful typing.

1. The Intent of a Program Description Language

A program description language serves traditionally the purpose of supporting the construction of programs, and helping with management tasks like version and configuration control (see e.g. [Win, Tie]). Tangram focusses on the first of these aspects, neglects the second one, and makes an attempt to support reusability instead. We will first discuss its intent of aiding the design of programs (or program parts), then we are going to discuss its potential role in supporting reusability. Finally we want to briefly digress to SETL in order to give an idea of the power of very high level constructs for program descriptions.

1.1 Aid in Designing Programs

The object oriented approach to program design with Ada, as sketched operationally in the Introduction, may be characterized by a combination of data and program abstraction. Following Wegner, *Data Abstraction* makes private types and operations on these types available to the user, while *Program Abstraction* goes beyond that by providing operations on objects whose representation and identity is hidden from the user ([Weg, p. 257]). Data abstraction is closely related to abstract data types, program abstraction to generic packages and their instantiations.

Booch's operationalization of the object oriented approach requires each object's implementation, after its views have been established. This requires the designer focussing again on the operations associated with each object. A program description language will be of substantial help in the design process if the connection between the objects and their operations is made tight in the following sense

- the operations (i.e. functions/procedures) are outlined on a functional level, holding the balance between going into too much detail, and superficially stating the mere intent of an operation,
- the application of data abstraction becomes visible, thus the use of abstract data types and their operations are indicated,
- the internals of this connection become visible (comparable to the construction of a finite state machine in which the hidden state and the state transitions which are hidden as well, are nevertheless specified).

The bare necessity of the underlying language requires stating the relation of entities in the package under consideration to other packages. This is done to ensure that entities like variables, constants, types and routines are exported from the proper package, and to specify what entities are made available by the present one. So this feature should be incorporated.

The treatment of data types and data structures may serve to illustrate the balance which has to be focussed on in designing such a description language. The data structuring facilities must not exercise too much generosity by not restricting the user too much (since the blueprint aimed at might be too sketchy to be useful), on the other hand it must not force the description of too many details (since in this case flexibility might be lost). Adding a second dimension, it should be possible to operate on different levels of specificity. Consider records as an example: it should be possible to operate with components of a record without being forced to name them (but using names should be fine, too), and it should be possible to completely specify a record, or to specify only the required components, which in turn should be allowed to contain type variables.

Using suitable transformational techniques, it should be possible to generate compilable Ada packages from Tangram descriptions. This goal is somewhat similar to the one pursued e.g. by the ESPRIT project SED (see e.g. [Kel]) using the prototyping language SETL. With this language used for specifications, one starts with a very high level prototype (which is very close to the formal specification, hence easy to verify), and gradually transforms this program into a functionally equivalent SETL program which is semantically on a much lower level. The transformations are correctness preserving, hence the resulting program is still correct. When further transformations within SETL are no longer profitable, one obtains a production efficient version of the program in Ada by a correctness preserving source-to-source transformation across the language boundary. This transformational approach is quite attractive, and we are just gaining some experience with it (now that the tools are constructed). It does not fit, however, into the object oriented paradigm to program construction. Let us point out some of the differences between the transformational, and the descriptive approach:

- a) transformations start from an executable prototype, descriptions are not executable,
- b) descriptions are oriented towards packages transformations start from whole programs (although transformations may be used technically on partial programs like e.g. SETL modules),
- c) descriptions use Ada's data structuring facilities for representing data (although not exclusively), transformations start from the repertoire of finite set theory (i.e. sets, maps, vectors), and select concrete data representations only in the very last step, when it comes to producing an Ada program,
- d) transformations are oriented towards functional behavior (hence tend to follow a more traditional functional approach), descriptions are oriented towards describing objects and their behavior.

Consequently, a translation of Tangram descriptions into compilable Ada packages will pursue other goals than the translation of SETL programs into Ada. This is essentially due to the fact that the translation SETL-Ada makes only sense if one assumes that the prototype is the complete solution to the given problem (albeit one that lacks the desirable performance) while the translation Tangram-Ada works from the more modest assumption that a blueprint describes part of the solution to a given problem.

1.2 Support Reusing Programs

Tangram may help in supporting reusability. Reusability of software is currently quite an active area of research in software engineering, and one of the main problems here is to being able to catch the meaning of a program, or program part, in order to retrieve programs by their functionality. The meaning of a program is difficult to characterize formally, and no practical way of describing it has yet been devised. *Formality* here means in particular that it must be possible with a reasonable amount of effort to do the following things:

- Describe the content of a piece of software in an understandable way (i.e. so that not only experts in e.g. λ -calculus are able to understand the description),
- Given the description of a desired functionality, retrieve from a depository of program fragments a piece which either has exactly the desired functionality, or which comes closest to it.

It is difficult to see that these requirements will ever receive a satisfactory solution; there are some approaches to the problem of reusability which focus on the problem of retrieving components (e.g. [PrFr] with the *functional decomposition*, or [MaKa] with an approach using *semantic* techniques). No approach known to this author deals in a satisfactory way with the problem of describing the meaning of a program fragment.

There are reasons to believe that reusability of software will most successfully be undertaken not on the level of the source code of a program fragment, but rather on the level of the concept that the program fragment is supposed to express. Cheatham notes: "The problem is that programs in any high level languages are the result of a mapping from some conceptual or abstract specification of what is to be accomplished into very specific data representations and algorithms which provide an *efficient* means for accomplishing the task at hand" ([Che, p. 589]). Hence one should be able to describe the concepts for an Ada package, say, in a suitable form, when it comes to address the question of reusability of the package. This conceptual description can be done at two different points in time: at *construction time*, when the mapping of objects and their operations to implementations is considered, and at *cataloging time*, when it comes to putting the package into a software depository for further use.

Using both the *a priori* and the *a posteriori* approach Tangram descriptions may support the process of characterizing software components. At construction time, the Tangram description of the package may be used as an approximation to a formal description of the package's content. At cataloging time the process of transforming a Tangram description into compilable Ada code, which has been sketched above, is reversed. Given an Ada package, one *extracts* a Tangram description which faithfully serves as its specification. This requires of course special skills (resembling the skills of a librarian who has to classify books for inclusion in a library - the analogy between searching for a book in a library and searching for a piece of software in a software depository has been emphasized in [PrFr]). Both the *a priori* and the *a posteriori* description are used then to characterize the package's content when it comes to search for a package with a specified functionality.

We will return to this aspect in Section 3.

1.3 Very High Level Constructs for Program Descriptions

It is obvious that programs written in a formalism close to formal specifications are easy to understand, and that they offer less obstacles to verification, than programs written in a formalism close to a machine. Very high level constructs may be used for such formalisms. These constructs are sometimes supported by or taken from an appropriate mathematical theory such as Horn clause logic (-PROLOG, see [Kow]), λ -calculus (-LISP, see [All]), array theory (-Nial, see [JGM]), and set theory (-SETL, see [SDOS]). The descriptive power of set theory for formulating algorithms has been described and convincingly demonstrated in [Kru]. We will focus on the latter one. Set theoretic constructs such as set, maps and vectors/tuples may be used for representing data; this together with the familiar control structures offers a rather natural way of expressing algorithms close to their formal specifications (which may be formulated mathematically using the same apparatus anyway). Consequently, we use in Tangram these constructs for a convenient formulation of the algorithms we want to represent. This means in particular that the process of data abstraction, i.e. the formulation of abstract data types (ADTs) is supported directly by these constructs. We will see how program abstraction is supported by this approach: the functional description of routines provides operations on the objects under consideration without revealing their representation in any detail.

Let us digress briefly and give an example of the expressive power of set theory as realized in SETL. This should give an idea of what we have in mind when it comes to concisely expressing algorithms. We want to compute the maximal cliques in an undirected graph $G = (V, E)$.

Definition. A subset $A \subseteq V$ is said to be complete iff $\forall x, y \in A - \{x\} : (x, y) \in E$ holds (hence no distinct vertices are connected by an edge). A is said to be a clique iff it is complete, and a maximal complete set (i.e. if $A \subseteq B$ implies $A = B$, provided B is complete).

If we want to compute all cliques in G in SETL, we may translate the mathematical description into code as follows; the vertices are assumed to be given as a set v , the edges as a set e each element of which is a set of two elements. The sets v and e are read in; the program prints the set of all cliques and is given in Fig. 1.

Its salient features are

- the explicit construction and use of sets (and nested sets, too, e.g. complete), and of set-theoretic constructs like the element- or the subset-relation,
- the use of assertions (if the expression after assert evaluates to false, the program aborts),
- the use of quantifiers (\forall, \exists),
- the omission of explicit variable declarations.

It is easy to see that this program is correct. This is so since it is nothing but a direct translation of the mathematical problem specification, where the relevant definitions have been expanded in a macro-like fashion.

Using set-theoretic constructs to describe blueprints for Ada packages will introduce some special problems, as far as data structures are concerned. We will use sets as usual in mathematics using

```

program AllCliques;
read (v, e);
$
$ check to see whether we read in the correct
  stuff
$
assert is-set(v);
assert is-set(e) and (  $\forall$  edge  $\in e \mid$  (edge  $\subseteq v$ 
and  $\#$  edge = 2));
$
$ compute the set of all complete subsets
$
Complete := {a:  $a \subseteq v \mid (\forall x \in a, y \in a - \{x\} \mid$ 
  (x, y)  $\in e$ )};
$
$ the cliques are the maximal sets in Complete
$
Cliques := {a:  $a \subseteq$  Complete  $\mid$  (not  $\exists b \subseteq$  complete
   $\mid a \subseteq b$  and  $a \neq b$ )};
print(Cliques);
end program AllCliques;

```

Fig. 1 SETL program for computing all cliques in a graph

value semantics: the following code

```

A := {1 ....10};
B := A;
A less := 10;

```

will not result in removing 10 from B. This works fine as long as no indirection is involved, but there are some difficulties as soon as the sets in question contain pointers. We will return to this problem in due course.

2. Language Description

This section will describe Tangram₁ in greater detail. We will first have a look at the overall organization of a Tangram₁ module, then we will discuss the language's type model. Roughly, a Tangram₁ module consists of a prelude section, and of its main chapter. These sections will be discussed together with the functional descriptions for the routines of the packages to be blueprinted. Finally, we will have to consider what a Tangram₁ module really means.

2.1 Organization of a Tangram Module

A Tangram₁ module is organized into a Prelude, and a section which we call the TVCR-Section. The Prelude establishes the visibility of outside and hidden objects to the module, and the TVCR-section (types, variables, constants, routines) indicates what is exported from this module, hence, what entities are made visible by it.

A module description will have to work with three different kinds of entities: with objects that are being made available from other modules, with objects that are being made available by the module itself, but which the module chooses not to reveal to the outside (hidden objects), and finally with abstract data types. ADTs play a special role here since they are not quite objects but rather templates.

As mentioned in the Introduction, Ada packages may be compared under the object oriented approach to state machines, which have some input and output, but which work with an invisible internal state, and in which the reaction to an input is determined by the input as well as by the internal state. Thus data are hidden as internal states of a package. On the other hand, the blueprint for a package must account somehow for the internal state, since otherwise pure functional descriptions would result. Thus the Prelude of a Tangram₁ module contains a provision for describing such an internal state.

The names introduced in the Prelude are visible throughout the module description, and in addition operations using ADTs are being made visible. For example, if a module uses the ADT set, and this ADT provides an operation called insert, then mentioning set as an ADT in the Prelude will make this operation visible (as insertset).

The TVCR-section contains the functional descriptions proper. Syntactically, types, variables, constants and routines are being made visible. For routines, we use Ada's syntax for their header lines (making names and signatures of the routines known to the environment, hence establishing visibility). For the routines involved,

we provide high level descriptions using constructs from finite set theory. These descriptions are local to each routine, in particular the names of the objects and entities used are local.

2.2 Tangram₁'s Type Model

This section is devoted to a brief discussion of the type system which is being used for Tangram₁. Since we want to blueprint applications using Tangram₁, we do not want to be too specific about the type of certain variables; this advocates using abstract data types, and type templates much in the spirit of SETL. On the other hand, we want to address entities like components of a record, say; here we have to be rather specific. Thus we have to have at our disposal Ada's types as well as SETL's.

This yields a curious blend. Suppose that A is a set of access variables each having a numeric component p. Suppose further that we take an arbitrary element x in A and increment x.p by 1. Note that we did not touch the set A, and that the old value of A is the same as the new value of this set (since the addresses did not change at all). But something has changed, and we have to account for the strange effects in the type system to be constructed.

In what follows, we refer to the grammar for Ada as given in [AJPO]. We need a reference point for the description of Ada's types, and here we choose the grammar above, but start with the axiom *type-declaration*. This yields a context free language which we denote by TYPE. In building up Tangram₁'s type system, we fix an at most countable set A of type variables such that the variables δ and ρ are no members of A, $G := (\delta, \rho)$, A is the base set we will be working with. Here δ stands for the discrete types in Ada, and ρ for the real ones. An interpretation connects this to TYPE as follows:

Definition. An interpretation of G is a map ϕ from (δ, ρ) into P (TYPE) such that each word in $\phi(\delta)$ derives from enumeration-type-definition, and each word in $\phi(\rho)$ derives from real-type-definition. A selection σ for an interpretation ϕ is a partial map σ from (δ, ρ) to TYPE such that $\sigma(t) \in \phi(t)$ holds for all $t \in \text{domain } \sigma$.

Interpretations describe intuitively what happens when enumerative and real types are elaborated; they are not really necessary in what follows since selections are so closely related: given an interpretation ϕ for G, we may reconstruct ϕ from its selections, since plainly

$$\phi(t) = \{\sigma(t); \sigma \text{ is a selection for } \phi\}$$

holds. Thus we will do without interpretations. We now describe type constructors over certain sets. Let H be a set of types.

Definition. The set $Rec(H)$ of all records over H is defined as

$$Rec(H) := \{(record, h) \mid h \in H^*\}$$

And, similarly, we define

$$Arr(H) := \{(array, i, e) \mid i \in \mathbb{N}^+, e \in H\}$$

is the set of all arrays with any number of elements from H , and

$$Acc(H) := \{(access, h) \mid h \in H\}$$

is the set of all pointers to objects from H .

Note that the set $Rec(H)$ is essentially the base set H^* of all sequences over H , but should not be identified with the latter set - otherwise it would be impossible to iteratively building up records.

Using the three constructors Rec , Arr and Acc , one builds up larger and larger sets. Now let M^* be the least fixed point for these constructs containing G (hence in particular the relations

$$Rec(M^*) \subseteq M^*, Arr(M^*) \subseteq M^*, Acc(M^*) \subseteq M^*,$$

hold, thus M^* is closed with respect to these type constructors).

For each of the constructors mentioned above, we are able to extend the interpretations and selections given for the base set. Because of the remark above, we will define only selections. These selections will map each of the sets constructed to M^* as follows: If the type constructor is Rec , then ϕ is a selection on $Rec(H)$ if given $(record, h) \in Rec(H)$, there exist selections ϕ_1, \dots, ϕ_k (with $k := \text{length}(h)$, $h = h_1 \dots h_k$), and maps ϕ_1, \dots, ϕ_k from H to the set of all identifiers such that

$$\begin{aligned} \phi(record, h) &= record \\ \phi_1(h_1) &:= \phi_1(h_1); \\ \phi_2(h_2) &:= \phi_2(h_2); \\ &\vdots \\ \phi_k(h_k) &:= \phi_k(h_k); \\ &\text{end record} \end{aligned}$$

holds. Similarly, if the type selector is Arr , then ϕ is a selection on $Arr(H)$ iff given $(array, i, e) \in Arr(H)$, there exist selections ϕ_1, \dots, ϕ_k on $\{e\}^+$ (with $k := \text{length}(i)$) and a selection τ on H such that

$$\phi(array, i, e) = array(\phi_1(i_1), \dots, \phi_k(i_k)) of \tau(e)$$

holds. We denote by $\tau(H)$ the set of all selections on H . Then: $(Acc(H))$ is defined through the one-to-one correspondence

$$\phi((access, h)) := access(\tau(h)),$$

where $\phi \in \tau(H)$.

This captures most of Ada's type system (we did not take care of variant records, or of task types). But in the same way we may incorporate SETL's type system here, and this is what we are about to do now. For this, let H be a set of types, then we define in a similar way

$$\begin{aligned} Set(H) &:= \{(set, g) \mid g \in H^*\}, \\ Tup(H) &:= \{(tuple, j) \mid j \in H^*\}, \\ Map(H, K) &:= \{(map, h, k) \mid h \in H, k \in K\}. \end{aligned}$$

Selections on these sets are defined in a rather canonical way: ϕ is a selection on $Set(H)$ with respect to a set α of selections on H iff

$$\phi(set, g_1 \dots g_k) = \{\phi_1(g_1), \dots, \phi_k(g_k)\}$$

holds for suitably chosen selections $\phi_1, \dots, \phi_k \in \alpha$. Analogously, we define selections on $Tup(H)$ relative to a set α of selections on H by $\phi(tuple, g) = \{\phi_1(g_1), \dots, \phi_k(g_k)\}$.

Denoting again the set of all selections on H relative to α by $\tau(H, \alpha)$, we define

$$\tau(Map(H, K), \alpha, \beta) := \{\tau(H, \alpha) \rightarrow \tau(K, \beta)\},$$

i.e. as the set of all maps from $\tau(H, \alpha)$ to $\tau(K, \beta)$.

Now let M^* be the least fixed point containing M^* with respect to the constructors Set , Tup , and Map , and denote by $\tau(M^*)$ the set of selections on M^* based on $\tau(M^*)$ as the set of basis selections. This is the type system we are working with. Thus a type in $Tangram_1$ may formally be considered as a member of $\tau(M^*)$. The salient features of this type system are that

- it incorporates Ada's as well as SETL's types, hence it is possible to move freely between the type systems of the two languages (keep in mind that we have excluded variant records as well as tasks, so the inclusion with respect to Ada's types is to be taken with a grain of salt),
- it allows formulating types that may have one or more types variables as components.

Equality may be formulated in this type system in such a way that e.g. two sets containing pointers may still be considered equal even if a value pointed at has changed in one set (provided they were equal before the change). This description is rather formal, however, and will be described in another paper.

2.3 The Prelude Section

This section serves a dual purpose by making visible all objects which are required in the TVCR-section. This applies to those entities that are defined externally (i.e. in other Tangram₁ modules) as well as to entities which are entirely private to the module under consideration. In addition the ADTs acted upon in the TVCR-section are dealt with here. Consequently, this section is subdivided into three parts, which syntactically are described as follows:

```
prelude
  Imports -- which entities are imported?
  IsADT -- which ADTs are required?
  Hidden -- which internally defined
        -- entities are visible?
end prelude;
```

The object oriented approach demands making the visibility of objects explicit, and this is one of the purposes of the Prelude section: whenever an object in the package under consideration needs to see an object from another module, this is the place to specify it. It may be somewhat surprising to see that hidden objects are made public, but the analogy with finite state machines with objects may help here: although states and state transitions of a finite automaton are not visible to an outside observer, it is nevertheless necessary to deal with them and to explicitly manipulate them. In the same sense it may be necessary to acknowledge that there are specific hidden objects in a package which need to be manipulated explicitly. An example indicating this may be helpful here: suppose that a package manipulating a text concordance has to be designed (see [Sol], Ch. 7). Then inserting and counting words from a text in this concordance requires partial knowledge of the concordance's structure.

Before describing these three components in greater detail, it should be mentioned that all names used in this section are visible throughout the package description which follows.

The Imports Section

Apart from ADTs this subsection lists all those items which are imported from other modules. This applies equally to constants, variables, types, and routines. Syntactically, we list first the package with its name, and the all entities as well as their properties which are imported from it. This is intended to clarify the visibility of each object (which in Ada proper is sometimes blurred by overusing use clauses). Name clashes, which may occur later will have to be resolved by qualification, but this is not important here. The syntax follows normal Tangram₁ conventions with the additional provision that roles are introduced into the description. A role is an informal characterization (by just one identifier) of an entity using the key word *ActAs*, as in e.g.

```
From OVER-USE: -- import from that package
Type t ... ActAs queue-buffer;
```

Hence the type *t* should be defined in the package OVER-USE. The explanation following *ActAs* does not have any formal, i.e. syntactical or semantical significance but may be thought of as an informal reminder of the role the type is intended to play. These roles are maintained in a separate dictionary which is associated with the package description.

Roles are not only associated with types but may also be attached to constants, variables and routines as well.

The IsADT Section

Here we make no ADTs visible. This is done by the keyword *IsADT* acting as an opener to this subsection, and a list of identifiers which are supposed to denote abstract data types. When the name of an abstract data type is listed here, it is assumed that a package description with this name exists. The names exported by such a package are then available in a qualified way. We will deal with these provider packages for ADTs later, but an example may be helpful here: Suppose that a package description says

```
IsADT Set, -- (*)
```

and that the Set package makes the following entities available:

```
Empty, -- the empty set
Intersection, Union, InsertElement,
DeleteElement,
QueryElement, InitializeEmpty -- usual
-- operations on sets
```

Then the specification (*) makes in addition to Set the objects *SetEmpty*, *SetIntersection* etc. available.

ADTs may be parametrized by a type or by another ADT (e.g. Queue (a) denoting queues with elements of type a). These ADTs are made available here, too; the parameter then may serve as a forward reference to the TVCR-section where the corresponding type is explained in greater detail. If one (or more) of these parameters is instantiated to a type already introduced (a known type), then another *less abstract* ADT is defined. This new ADT inherits all operations from its parent ADT. Now this is indicated syntactically may be seen from the example displayed in Fig. 2.

The Hidden Section

In terms of the entities already visible which are either imported or come from ADTs it is usually necessary to make some internal objects available. Since packages may be assumed to work on some hidden internal state, an explanation of a package's internal working should to some extent be based on the knowledge of that state. This is what happens here. The analogy to the private part of a package specification of an Ada package comes to mind. The private part reveals the implementation of a data structure as far as necessary for type checking. The objects declared here as hidden reveal in a similar way their internal structure only to the extent which is required by the context (i.e. to make names visible). Hence both parts are somewhat similar indeed, although the similarity comes from different motivations.

2.4 The TVCR-Section

After having outlined what entities are imported from the package under consideration - hereby establishing passive visibility - it becomes necessary to establish active visibility. This serves the dual purpose of completely establishing the visibility for each object, and of outlining the interface together with a functional description for each object.

This section tells the world outside which types are exported, where the *semantics* of the types involved follows the outline given above. Hence a blend of Ada's and SETL's types may be made available by a package. This implies in particular that a `Tangram` type may contain type variables. Substituting all type variables for types proper evidently correspond to instantiating a generic type, but it is possible as well that only a partial substitution is done when it comes to use the package within the `Tangram` system. This would correspond to a partial instantiation and does not have a formal counterpart in Ada. *Syntactically* we follow here much of Ada's syntax with

some modifications. These modifications address the handling of records. If we say e.g. that

```
type tau is record
    integer; real;
end record;
```

then it is implied that among the components of the record type representing tau at implementation time there will be a component of type integer, and a component of type real, respectively. If v is a variable of type tau, then v.#1 denotes the component of type integer, and v.#2 denotes the component of type real, resp. Instantiating this record template will take care of consistent naming. In this way we hope to at least partially bypass Ada's restrictions with respect to records as generic parameters.

This section contains, too, those type definitions which have to be filled in because of forward references in the `IsADT` section.

The declaration of variables and of constants follows rather the same pattern as the corresponding declarations in Ada do. Here it should be noted that variables may be parametrized implicitly according to type variables that occur as components of their types, and that defining a constant implies no type variables being involved in the corresponding entities.

This section contains the signatures of the routines implemented by and exported from the package. Hence we state in this section the routines together with their respective kind (i.e. function or procedure), the names, modes and types of their parameters as well as the type of the value returned, if we are dealing with a function. This header line is formulated in much the same way as in Ada, and it serves as an opener to the routine's functional description, in which a high level outline of the algorithmic content is given. This outline is presented in a manner resembling SETL's diction of expressing algorithms.

We will discuss details below, but before doing this, we want to present an example which based on Booch's text concordance problem (see [Bo], Ch. 7 for details). For the sake of brevity we focus on one particular routine - the procedure `add`. The `Tangram` description is given in Fig. 2.

Thus we import two types from other packages (for which we assume that there are package descriptions available), and make the abstract data types `set` and `queue` available. These ADTs are parametrized (the types of the elements serve as parameters), they are instantiated to the ADTs `queue1`, and `set1`, respectively. This implies the availability of the operations on

these types as discussed above. The package has a hidden anonymous type and another hidden object, viz., a set x with elements taken from that type. All this is declared in the prelude section. The procedure `add` takes a word and a line number and adds either the line number to a queue of line numbers, or it adds the word together with an initialized queue to the set x . If the set is full, an exception is raised.

Exceptions have to be made visible in much the same way as e.g. types or routines. This section is the proper place for doing that. In the same way as ANNA we distinguish three ways of describing an exception. The first way the *default* way: it is just stated that an exception is raised. If the conditions are explicitly stated under which conditions an exception is raised, then we have a *weak* description of that exception. If finally the package's internal state is described immediately before the exception is raised, we consider this a *strong* description (see [LKHO, p. 99]).

The package description may serve as a very high level draft of the package - it should be evident from the example above that this may be useful during the very first steps of the design of a package. Alternatively, the description may serve to concisely describe the algorithmic content of a package for reusing it. The functional description is essential here, and we will discuss it in a moment. The example also shows that it should not be too difficult to generate executable Ada code from the package description, given the experiences with generating efficient Ada code for the version of finite set theory represented in SETL (see [DoGu]).

```
Package description CONCORDANCE is
prelude
  imports
    package WORDS:
      type word is string;
    package LINE-NUMBERS:
      type number is positive;
  is ADT set(g), queue(h); -- ADTs parametrized
  hidden
    instantiate queue1 as queue (h => LINE-NUMBERS.
                                number);

    instantiate set1 as set (g => record WORDS.word;
                           queue 1; end record);

    -- This makes all operations on the ADTs
    -- queue(h), and set(g), resp.,
    -- available in an appropriate way

  x: set1;
end prelude;
```

```
procedure add(the-word: in WORDS.word;
              the-number: in LINE-NUMBERS.num-
                           ber);

functional
  if exists k in x such-that k.#1 = the-word
  then insert$queue1(k.#2, the-number);
  else insert$set1(x, (the-word, initialize$
                     queue1(the-number)));
    when is-full$set1 => raise overflow;
  fi;
end functional;
end description CONCORDANCE;
```

Fig. 2 Tangram description for Booch's CONCORDANCE problem

2.5 Functional Descriptions

Prelude and header lines in a package description are quite oriented towards Ada (since this is the target language anyway). The functional description, however, should be concise and of a very high level. Hence it is difficult to use Ada here, since such operations as iterating over a set and the like are evidently not available as built-in operations in Ada. On the other hand, set theory provides a very natural way of expressing algorithms. This is so because e.g. sets, maps and relations are easily used to describe the combinatorial structure underlying all algorithms. Consequently, it is our hypothesis that finite set theory is an adequate vehicle for the program development process. This is demonstrated by the SETL programming language. SETL, however, does not fit directly on top of Ada as a program description language because it is quite easy to lose the link between a SETL structure, and the corresponding Ada structure which is intended to represent it - and vice versa. Thus there should be a descriptive level between Ada as the language to implement an algorithm and SETL (or set theory) as the language to describe the algorithm. This descriptive level is provided by Tangram, or, to be more specific, by the functional descriptions outlining what we have called *algorithmic content* above.

We borrow from SETL those constructs which deal with sets, maps and tuples. The discussion of the type structure above shows that Ada structures may be contained in these set theoretic entities (so that we may have a set of records, or a map from pointer variables to arrays). As far as notation is concerned, we use the familiar mathematical notation for sets and tuples. Hence

```
(e(x-1, ..., x-k):x-lcc-1, ..., x-kcc-k |
  P(x-1, ..., x-k))
```

denotes the set of all objects $e(x_1, \dots, x_k)$ such that x_i is taken from C_i , $1 \leq i \leq k$, with the property that the predicate $P(x_1, \dots, x_k)$ holds. Tangram₁ provides the usual operations on sets (union, intersection, set difference, power set, inserting and deleting elements, testing membership, subset relation etc.), and on tuples (slicing, concatenation, indexing, to name just a few) in the same way as mathematical set theory does. Maps are the usual associative structures and may be used to retrieve image values.

The question may arise here why Tangram₁ provides set theoretic constructs, but allows sets as abstract data types to be formulated. To see why this makes sense, we have to point out which role is played by the different kinds of sets. When formulated as ADTs, sets are used as implementation structures, so these sets will have to be represented explicitly in the computer's memory. When used as descriptive structures, sets are used as mathematical entities which not necessarily have to be constructed explicitly, but rather may be transformed out, i.e. which may be substituted after a suitably chosen transformational process by simpler implementation structures.

It is possible in Tangram₁ to iterate implicitly over a compound structure with the existential and the universal quantifier, respectively; existential and universal quantification both yield Boolean values, the former one producing additionally the existing value if it returns TRUE.

In a similar way we make the usual statements of a procedural language available. This applies to

- conditional statements
(if ... then ... elsif ... then ... else ... fi)
- case statements
- iterative statements (simple loops, for-, while- and until- loops). These constructs may be left using a quit-statement, iteration may skip a value using a continue-statement.

We want to emphasize the following statements which may help arguing about descriptions. The assert-statement takes a Boolean value as an argument (e.g., `assert x ≥ 0`) and may consequently be used to conveniently formulate preconditions. An achieve-statement also takes a Boolean value as an argument and may serve as a postcondition, e.g.

```
achieve v[i] (1 ... #t-1) such-that t[i] < t[i+1]
```

makes sure that the (dynamic) tuple t is sorted. The achieve-statement is imperative in nature: saying achieve A means that an algorithm has to be devised that makes the condition A true (in contrast to assert A

which only states that A must be true). The select-statement allows to non-deterministically select an entity from a compound object like a set, map or tuple. For example

```
select xct such-that v yct: x > y
```

selects the maximal member of t , where t may be a set or a tuple. In analogy to the achieve-statement we think of the select as imperative in nature, i.e. an algorithm has to be found which takes care of the selection. This fits to the intent of Tangram₁ as a program description language allowing the formulation algorithms on a very high level without going into too elaborate details.

2.6 Provider-Packages

These packages are intended to define abstract data types, thus they provide a service (rather than consume a service like an actor-package). In a provider package the imports clause in the prelude section is empty, since an ADT is perceived as an independent, basic and thus axiomatic mathematical entity. Thus ADTs must not depend on other entities. On the other hand, it does not violate this perception of ADTs as axiomatic entities that the hidden clause in the prelude is not empty, and the `isADT` clause may be present, too, since an ADT may be incorporated by another one. The TCVR-section lists the name of the ADT together with its parameters. This name is qualified as an ADT by using `ADT` instead of `type`, which would be used for characterizing type names in non-provider packages. The functional descriptions and all other constructs remain unchanged. Fig. 3 displays the package description for a provider package which makes the ADT `set(s)` available. For the sake of simplicity we display only the definition of the operator `*` for intersecting sets. Note the use of the assertion that every element of the sets involved is of the right type; Tangram₁ provides a type checking function (which is of an assertive nature).

As indicated above, mentioning an abstract data type in the prelude of a package description makes all abstract operations on the data type available.

```

package description Set
prelude
end prelude;

ADT set(a);
function "+"(x: in set(a); y: in set(a))
    return set(a);

functional
    assert forall w in x such-that type-of(w) = a;
    assert forall w in y such-that type-of(w) = a;
    achieve z = (b in x such-that b in y);
    return z;
end functional;
end package description;

```

Fig. 3 A Tangram₁ provider Package

3) The Tangram System

This section provides a brief overview over the Tangram system. As the name indicates, this system is intended to maintain a set of packages as building blocks from which programs may be composed. The system is not yet fully implemented.

A particular package is represented by its Tangram₁ description; this description is intended to adequately express the package's intent. This is not the only aspect of a package which is stored in the Tangram system. Tangram collects different views of a package, and Fig. 4 indicates which views are relevant here. Starting counterclockwise from the *Code* node, we will briefly discuss each component now.

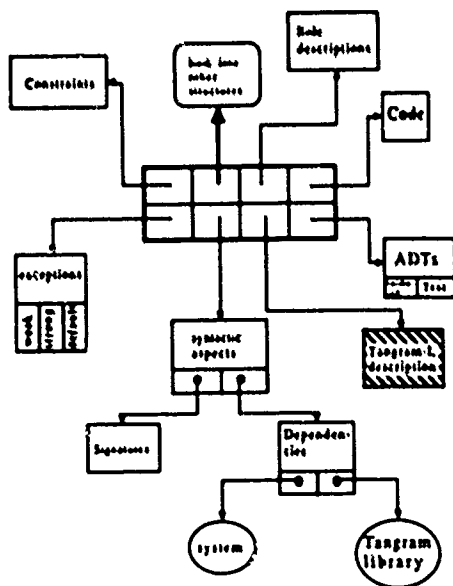


Fig. 4 An overview of the Tangram system

The Ada code for the package is stored in the node labelled *Code*; we store here not the textual representation itself but rather the corresponding abstract syntax tree - this makes manipulations on the code easier. The next node contains role descriptions, i.e. pointers into the code where entities are annotated using ActAs. The node labelled *Constraints* indicates which restrictions are imposed on the package (e.g. which operating system, which implementation restrictions, what storage capacity is required internally as well as externally etc.) These restrictions are formulated in plain English and given in tabular form. The next main slot *Exceptions* is occupied by a list of exceptions, which in turn are categorized according to *weak*, *strong* and *default* annotations, as described above. Syntactic aspects are stored in the next node, and here we focus on

- the signatures of procedures and functions, i.e. the list of formal parameters together with each parameter's modes,
- the dependency graph for the package: this graph indicates on which other packages the package under consideration depends. These dependencies are categorized according to packages provided by the Ada system under which the package is intended to run, and user defined packages which are maintained by Tangram.

The next slot is occupied by a pointer to the Tangram₁ description as discussed in the present paper. Finally we find a node labelled *ADTs*; here we maintain information about the abstract data types which are used by the package under consideration. This information is presented in textual form for enabling the user to determine what the algebraic characteristics of the ADT are; an animated representation similar to that presented in the Salsa-II system (see [Bro]) but using animation techniques is under construction.

Tangram₁ programs may obviously not be used for generating executable code on any real machine. The Tangram₁ compiler is rather used for

- consistency checking: it is checked that importing and exporting objects - hence making objects visible - is done properly and consistently.
- filling in slots: most of the slots in a node for a package in the Tangram system may be generated automatically. If only a Tangram₁ description for a package is available, then these slots may be filled from the knowledge available through this description.

In addition, Tangram₁ descriptions will serve as an input to the transformational engine that derives Ada programs from a description. This will have to be done in a very similar way as in [DoGu], although some sort of knowledge base for representing ADTs and the algorithms manipulating them will have to be added.

Within the research related to Tangram, we concentrate first on the description of individual packages, since we feel that this information is central to reusability. We have not decided yet how to integrate these descriptions into a system which supports retrieving individual packages. These are at least the following alternatives to consider:

- the *faceted library scheme* due to Prieto-Diaz and Freeman [PrFr] which seems to be a fairly practical approach, although it appears to be rather awkward to add new categories to this scheme (in particular to the weighted graph of concepts),
- a scheme using conceptual clustering, as e.g. Maarek and Kaiser [MaKa] propose. This is a very promising approach, although it appears to have been tried out only according to a syntactic categorization, rather than on semantic categories.
- an approach oriented towards ADTs coupled with an expert system (quite comparable to the SESMO₀ system developed at GMD Karlsruhe in conjunction with IBM Germany, see [Zim]) - here it appears that one would need a domain to argue about which is somewhat more general than ADTs.
- an approach using normal forms, as proposed by Luqi and Ketabchi [Luq].

A combination of conceptual clustering, normal forms and hypertext techniques (for linking the individual nodes to each other in a sophisticated way, see e.g. [Con]) seems to be the most promising approach and will be pursued further.

4. Conclusion

We have discussed the construction of a program description language, and have seen how this language may fit into the object oriented approach to software development with Ada. The approach to constructing this language centered around the paradigm of developing programs using a very high level language based on set theory (much like SETL). Finally we have indicated how the Tangram₁ descriptions fit into an attempt to overall describing the functionality of an Ada package. It will remain to be seen how this approach may be utilized in an integrated system

supporting reusability.

We have pointed out that from a Tangram₁ description an Ada program may be generated - at least in principle. Further research will show how and to what extent the practical experiences gained with translating SETL to Ada may be capitalized upon.

Ernst-E. Doberkat:

Currently a full professor for software engineering in the Department of Mathematics at the University of Essen, from 1985 through 1988 Chairman of the Computer Science Department at the University of Hildesheim, 1981 - 1984 Associate Professor of Mathematics and Computer Science, Clarkson College of Technology, Potsdam, N.Y.

Research interests: program transformations, reusability of software, prototyping, semantics of programming languages, analysis of algorithms



References

- [AJPO] The Programming Language Ada Reference Manual. *Lecture Notes in Computer Science*, vol. 155, Springer-Verlag, Berlin, 1983
- [All] Allen, J.: Anatomy of LISP. McGraw-Hill Book Company, New York, 1978
- [Bo1] Booch, G.: Software Engineering with Ada. Benjamin/Cummings, Menlo Park, 1986
- [Bo2] Booch, G.: Software Components with Ada - Structures, Tools, and Subsystems, Benjamin/Cummings, Menlo Park, 1987
- [Bro] Brown, M.H.: Exploring Algorithms Using Balsa-II. *IEEE Computer*, May 1988, 14 - 36
- [Che] Cheatham, T.E.: Reusability Through Program Transformations. *IEEE Trans. Softw. Eng.*, vol. SE-10, 1984, 589 - 594
- [Con] Conklin, J.: A Survey of Hypertext. Technical Report STP-356-86, Rev. 2, Software Technology Program, MCC, Austin, Texas, Dec. 1987
- [DoGu] Doberkat, E.-E., Gutenbeil, U.: SETL to Ada - Tree Transformations Applied. *Information and Software Technology* 29, 10 (1987), 548 - 557
- [Mil] Hill, A.: The Formal Specification and Verification of Reusable Software Components Using Ada with Asphodel. *Ada User* 9, June 1988, 113 - 123
- [JGM] Jenkins, M.A., Glasgow, J.I., McCrosky, C.D.: Programming Styles in Nial. *IEEE Software*, Jan. 1986, 46 - 55
- [Kel] Keller, J.P. et al (Eds.): The SED Approach to Program Construction. Forthcoming Report in the Springer ESPRIT series, 1989
- [Kow] Kowalski, R.A.: Logic as a Computer Language. K.L. Clark, S.-A. Tärnlund (Eds.): *Logic Programming*, Academic Press, London, 1982, 3 - 18
- [Kru] Kruchten, Ph., Schonberg, E., Schwartz, J.: Software Prototyping Using the SETL Programming Language. *IEEE Software*, Oct. 1984, 66 - 75
- [Lev] Levy, H.M.: Capability-Based Computer Systems. Digital Press, Bedford, 1984
- [LHKO] Luckham, D.C., v. Henke, F.W., Krieg-Brückner, B., Owe, O.: ANNA - A Language for Annotating Ada Programs. *Lecture Notes in Computer Science*, vol. 260, Springer-Verlag, Berlin, 1987
- [Luq] Luqi and Ketabchi, M.: A Computer-Aided Prototyping System. *IEEE Software*, March 1988, 66 - 72
- [MaKa] Maarek, Y.S., Kaiser, G.E.: Using conceptual clustering for classifying reusable Ada code. *Proc. ACM SIGAda International Conference, ACM Ada Letters*, Dec. 1987, 208 - 215
- [Pai] Paige, R.: Programming with Invariants. *IEEE Software*, Feb. 1986, 56 - 69
- [PrFr] Prieto-Diaz, R., Freeman, P.: Classifying Software for Reusability. *IEEE Software*, Jan. 1987, 6 - 16
- [SDDS] Schwartz, J., Dewar, R., Dubinsky, E., Schonberg, E.: Programming with Sets - An Introduction to SETL. Springer-Verlag, New York, 1986
- [TIC] Tichy, W.F.: Tools for Software Configuration Management. In [Win], 1 - 20
- [Wag] Wagner, P.: On the Unification of Data and Program Abstractions in Ada. *Proc. 10th Ann. ACM Symp. on Princ. of Progr. Lang.*, 1983, 256 - 264
- [Win] Winkler, J.F.H. (Ed.): International Workshop on Software Version and Configuration Control. B.G. Teubner, Stuttgart, 1988
- [Zim] Zimbel, R.: SESMOD - Ein Expertensystem zur Auswahl von wiederverwendbaren Programm-Modulen. Technical Report, Forschungszentrum Informatik, Karlsruhe, September 1988

AdaL, An Automated Code Reuse System

George C. Harrison

Norfolk State University, Norfolk Virginia

Under a United States Army grant we are developing a prototype software package to produce an automated Ada code reuse system supported by the language LIL to aid the Ada programmer/designer in choosing the appropriate Ada generic or ordinary package from a data base of reusable code and to automatically instantiate that code if it is generic. Our goal is to have reusable code chosen effectively WITHOUT actually examining Ada specifications. By examining the semantics in the LIL files the programmer may choose the appropriate LIL file that corresponds to the specifications and semantics needed in his or her Ada source.

INTRODUCTION

The notion of reusing software, especially source code, has become an established practice in the United States. The tools to aid in this practice have not been in wide use although the development of such methodologies are attracting the attention of many researchers. Investigators differ in their approaches to developing such tools, but there is a common thread of agreement that efficient software should exist to aid the development, evaluation, testing, storage, and integration of source code libraries.^{2,5}

Our investigations are concentrating on the integration of reusable software into ongoing projects.³ We have been most interested in the reuse of Ada source code and have used Ada as the primary development tool for our user interfaces to the integration of the stored source code. This was done out of the demand to "prove" that Ada can be used effectively as reusable components, out of the goals of the supporting research contract, and a bit out of our zeal to defend Ada's qualities.

Our work has been sponsored primarily through a sizable research and equipment

grant from the United States Army (ARO Proposal # 25510-EL-M) by way of the Army Research Office and AIRMICS.

REUSE INTEGRATION

For projects using Ada the fostering and utilization of reusable packages should be of a primary concern and should be a common practice; however, if it is not easy to find information about the location, kinds and utility of the source code available then they probably should not have been written. Yet another problem arises when there is a concern about the correctness and the applicability of the code to the overall design.

We have also found some frustrations in searching for the right piece of code for our applications. By scrutinizing a generic package we should of course find the formal parameters, the necessary visible data types, the operations, and the exceptions. Even if the code is well documented it is often unclear what the formal parameters to the generic package represent and what the limitations to the semantics in the package body are.

As an extreme example look at this generic complex operations package:

```
-----  
generic  
  type BASE is private;  
  with function PLUS(B1, B2 : in BASE)  
    return BASE;  
  with function MINUS(B1, B2 : in BASE)  
    return BASE;  
  with function TIMES(B1, B2 : in BASE)  
    return BASE;  
  with function DIVIDE(B1, B2 : in BASE)  
    return BASE;
```

package CMX_OPERATIONS is

type CMX is

record

 REA : BASE;

 IMA : BASE;

end record;

function "-"(Z : in CMX) return CMX;

function CONJUGATE(Z : in CMX)

 return CMX;

function CMX_TYPE(R : in BASE)

 return CMX;

function REA_PART(Z : in CMX)

 return BASE;

function IMA_PART(Z : in CMX)

 return BASE;

function "+"(Z1, Z2 : in CMX)

 return CMX;

function "*" (Z : in CMX;

 F : in BASE) return CMX;

function "-"(F : in BASE;

 Z : in CMX) return CMX;

function "-"(Z1, Z2 : in CMX)

 return CMX;

function "-"(Z : in CMX;

 F : in BASE) return CMX;

function "-"(F : in BASE;

 Z : in CMX) return CMX;

function "+"(Z1, Z2 : in CMX)

 return CMX;

function "+"(Z : in CMX;

 F : in BASE) return CMX;

function "+"(F : in BASE;

 Z : in CMX) return CMX;

function "/"(Z1, Z2 : in CMX)

 return CMX;

function "/"(Z : in CMX;

 F : in BASE) return CMX;

function "/"(F : in BASE;

 Z : in CMX) return CMX;

function "***"(Z : in CMX;

 I : in INTEGER)

 return CMX;

ZERO_DIVIDE : exception;

-- raised when divisor is zero element

-- in "/" functions and raised when Z

-- is zero element and I is negative

-- in the "***" function

procedure GET(Z : out CMX);

procedure PUT(Z : in CMX);

end CMX_OPERATIONS;

A mathematician might receive a lot of enjoyment manipulating this package by repeatedly instantiating BASE with arbitrary data types and instantiating PLUS, MINUS, TIMES and DIVIDE by any four binary operators. Normally of course such a package would only allow some implementation of floating point numbers, but the package above might be used to work on lattice points or some mathematical applications where BASE might represent or-

dinary complex numbers or the data type of 4 by 4 matrices over the integers.

We may not know that the implementation might make assumptions about the BASE data type that could either cause run-time errors or erroneous results. For example somewhere in the implementation there might be an assumption that the data type has a 'MULTIPLY' inverse and identity: MULTIPLY(B1, INVERSE_B1) = identity, that MINUS(MULTIPLY(B1,B2), MULTIPLY(B2,B1)) = the 'ADD' identity, or that there is a precedence of operators.

Ada has no real way of guarding against bad assumptions or against suppositions about the axiomatic necessities of the data types, operators, etc. in an instantiation. Thus, we are concerned that there might be misinterpretations of the actions of an implementation in a reuse library.

Our initial answer to this problem was to try to use a form of LIL (Library Interconnection Language) that provides and extends many Ada semantic ideas: Theories, Views, Generics, etc.1,3 Although we are still in the process of refining the syntax of our form of LIL we have developed a prototype of a software tool, Adal, that is primarily a user interface to using LIL to support the reusable qualities of source code.

THE LIL SPECIFICATIONS

Normally, any implementation of Adal would have provided a hierarchical collection of data theories that could be both modified and added to as the need arises. For example to "insure" a correct use of the generic complex operations package above Adal might provide the following LIL structures:

theory TRIVIAL_SET is

 type ELT;

end TRIVIAL_SET;

generic

 type ELT is TRIVIAL_SET;

theory BINARY_OPERATION is

 function BI_OP (E1, E2 : ELT)

 return ELT;

end BINARY_OPERATION;

```

generic
  type M is TRIVIAL_SET;
  function "*" is BINARY_OPERATION;
theory MONOID is
  function "*" ( M1, M2 : M ) return M;
  (assoc, id:1);
end MONOID;

*****

generic
  type M is MONOID;
theory GROUP is
  function "*" ( M1, M2 : M ) return M;
  (assoc, inv, id:1);
end GROUP;

*****

generic
  type G is GROUP;
theory ABELIAN_GROUP is
  function "*" ( G1, G2 : G ) return G;
  (assoc, comm, inv, id:1);
end ABELIAN_GROUP;

*****

generic
  type ELT is TRIVIAL_SET;
theory SINGLE_VARIABLE_FUNCTION is
  function FTM ( E1 : ELT ) return ELT;
end SINGLE_VARIABLE_FUNCTION;

*****

generic
  type R is ABELIAN_GROUP;
  function m is
    SINGLE_VARIABLE_FUNCTION;
  function "-" is BINARY_OPERATION;
  function "*" is BINARY_OPERATION;
theory RING is
  generic
    type R is GROUP;
  view Aplus
    ( RING => ABELIAN_GROUP (R) ) is
    ops ( "-" => "-" );
  end Aplus;
  --
  -- Because of the traditional view of
  -- using + instead of * as the
  -- operator in the underlying abelian
  -- group in a ring we use this generic
  -- view to change its notation before
  -- using * as the multiplicative RING
  -- operator

  function "+" ( R1, R2 : R ) return R;
  (assoc, comm, inv, id:0);
  function m ( R1 : R ) return R;
  function "-" ( R1, R2 : R ) return R;
  function "*" ( R1, R2 : R ) return R;
  (assoc);

  var R1, R2, R3 : R;

```

```

axioms
  (right_distributive_*/+ ::=
    ((R1 + R2)*R3 =
      (R1*R3) + (R2*R3)) );
  (left_distributive_*/+ ::=
    (R1*(R2 + R3) =
      (R1*R2) + (R1*R3)) );
  ( m(R1) = R1inv );
  ( R1 - R2 = R1 + m(R2) );

end RING;

*****

The programmer now would be in a position
to write a LIL generic abstraction
for GENERIC_COMPLEX_OPERATIONS:

*****

generic
  type COMPONENT is RING;
  function "+" is BINARY_OPERATION;
  function "-" is BINARY_OPERATION;
  function "*" is BINARY_OPERATION;
  function "/" is BINARY_OPERATION;

package GENERIC_COMPLEX_OPERATIONS is

  type COMPLEX is record
    REAL : COMPONENT;
    IMAGINARY : COMPONENT;
  end record;

  function "-"(Z : in COMPLEX)
    return COMPLEX;
  function CONJUGATE(Z : in COMPLEX)
    return COMPLEX;
  function COMPLEX_TYPE(R : in COMPONENT)
    return COMPLEX;
  function REAL_PART(Z : in COMPLEX)
    return COMPONENT;
  function IMAGINARY_PART(Z : in COMPLEX)
    return COMPONENT;
  function "+"(Z1, Z2 : in COMPLEX)
    return COMPLEX;
  function "+"(Z : in COMPLEX;
    F : in COMPONENT) return COMPLEX;
  function "+"(F : in COMPONENT;
    Z : in COMPLEX) return COMPLEX;
  function "-"(Z1, Z2 : in COMPLEX)
    return COMPLEX;
  function "-"(Z : in COMPLEX;
    F : in COMPONENT) return COMPLEX;
  function "-"(F : in COMPONENT;
    Z : in COMPLEX) return COMPLEX;
  function "*" (Z1, Z2 : in COMPLEX)
    return COMPLEX;
  function "*" (Z : in COMPLEX;
    F : in COMPONENT) return COMPLEX;
  function "*" (F : in COMPONENT;
    Z : in COMPLEX) return COMPLEX;
  function "/" (Z1, Z2 : in COMPLEX)
    return COMPLEX;
  function "/" (Z : in COMPLEX;
    F : in COMPONENT) return COMPLEX;
  function "/" (F : in COMPONENT;
    Z : in COMPLEX) return COMPLEX;

```

```

function ""(Z : in COMPLEX;
  I : in INTEGER) return COMPLEX;

procedure GET(Z : out COMPLEX);
procedure PUT(Z : in COMPLEX);

exception
  DIVISION_BY_ZERO;

var C : COMPLEX;
  I : INTEGER;

axioms
  (C / (0,0) = DIVISION_BY_ZERO);
  (C = (0,0) and I (INTEGER 0 =>
    POWER(C,I) = DIVISION_BY_ZERO);
  precedence
    0: COMPONENT;
    1: *COMPONENT, /COMPONENT;
    2: *COMPONENT, -COMPONENT;);

and GENERIC_COMPLEX_OPERATIONS;

```

.....

If the programmer has the following Ada specification and body, we will be able to make a "correct" instantiation:

```

-----
package DATA_TYPE is

  type S7 is new INTEGER range 0 .. 6;

  function SUM(S1, S2 : in S7) return S7;
  function DIFFERENCE(S1, S2 : in S7)
    return S7;
  function PRODUCT(S1, S2 : in S7)
    return S7;
  function QUOTIENT(S1, S2 : in S7)
    return S7;

end DATA_TYPE;

package body DATA_TYPE is

  function SUM(S1, S2 : in S7) return S7
  is
  begin
    return S7((INTEGER(S1) + INTEGER(S2))
      mod 7);
  end SUM;

  function DIFFERENCE(S1, S2 : in S7)
    return S7 is
  begin
    return S7((INTEGER(S1) - INTEGER(S2))
      mod 7);
  end DIFFERENCE;

  function PRODUCT(S1, S2 : in S7)
    return S7 is
  begin
    return S7(INTEGER(S1) * INTEGER(S2)
      mod 7);
  end PRODUCT;

```

```

function QUOTIENT(S1, S2 :
  in S7) return S7 is
begin
  case S2 is
    when 0 => raise NUMERIC_ERROR;
    when 1 => return S1;
    when 2 =>
      return S7(INTEGER(S1)*4 mod 7);
    when 3 =>
      return S7(INTEGER(S1)*5 mod 7);
    when 4 =>
      return S7(INTEGER(S1)*2 mod 7);
    when 5 =>
      return S7(INTEGER(S1)*3 mod 7);
    when 6 =>
      return S7(INTEGER(S1)*6 mod 7);
  end case;
end QUOTIENT;

end DATA_TYPE;

```

The use of views allows us to justify how a given LIL entity satisfies a given LIL theory; these also form the fundamental communication links between LIL entities and Ada compilation units:

.....

```

-- with Ada DATA_TYPE
view RING_BASE : RING => S7;
types ( R => S7 );
ops ( "+" => ADD );
  ( "-" => SUBTRACT(0, ) );
  ( "-" => SUBTRACT );
  ( "*" => MULTIPLY );
end RING_BASE;

-- with Ada DATA_TYPE
view PL : BINARY_OPERATION => SUM;
types ( ELT => S7 );
ops ( BI_OP => SUM );
end PL;

-- with Ada DATA_TYPE
view MI : BINARY_OPERATION => DIFFERENCE;
types ( ELT => S7 );
ops ( BI_OP => DIFFERENCE );
end MI;

-- with Ada DATA_TYPE
view TI : BINARY_OPERATION => PRODUCT;
types ( ELT => S7 );
ops ( BI_OP => PRODUCT );
end TI;

-- with Ada DATA_TYPE
view DI : BINARY_OPERATION => QUOTIENT;
types ( ELT => S7 );
ops ( BI_OP => QUOTIENT );
end DI;

```

.....

Note that S7 is actually more than an ordinary ring: it's a finite field.

'View packages' allow a named association between LIL and Ada packages:

.....

```
view package GENERIC_COMPLEX_OPERATIONS
  = CHX_OPERATIONS
  types ( COMPONENT => BASE );
  ops   ( "+" => PLUS );
        ( "-" => MINUS );
        ( "*" => TIMES );
        ( "/" => DIVIDE );
end GENERIC_COMPLEX_OPERATIONS;
```

.....

The use of the 'make' listing allows us to make a 'correct' and automated Ada instantiation. Thus, using the above views

.....

```
make COMPLEX_OPERATIONS is new
  GENERIC_COMPLEX_OPERATIONS
  ( COMPONENT => RING_BASE,
    "+" => PL,
    "-" => MI,
    "*" => TI,
    "/" => DI ); end;
```

.....

automatically produces the following Ada instantiation:

```
-- with LIL make COMPLEX_OPERATIONS
package COMPLEX_OPERATIONS is new
  CHX_OPERATIONS
  ( BASE => S7,
    PLUS => SUM,
    MINUS => DIFFERENCE,
    TIMES => PRODUCE,
    DIVIDE => QUOTIENT );
```

This may seem to be a lot of work for one simple instantiation; however, there are three items to remember: A. Theories only have to be built once; the more theories implemented the fewer have to be constructed for new software. B. Although careful LIL packages have to be constructed to match the specifications AND the semantics of the corresponding Ada packages, they and their corresponding view links to Ada become as permanent as the Ada reusable code. C. The work in building views and the corresponding 'make' to use the Ada source 'correctly' should be well worth the effort in system integration.

THE TOOL

AdAL is a software tool written in VAX Ada using Digital Equipment Corporation's Run Time Library (RTL) of Screen Management Guidelines (SMG) to facilitate the user interface. Fundamentally, it's a subsystem to VMS that allows the user to work with the reusable library, interface with VAX Ada commands, etc. We have only utilized the VMS v4.7 SMG routines that are compatible with v5.0 SMG routines to maintain compatibility with most current VAX/VMS systems. We have attempted compatibility with all VT100, VT200, etc. terminals.

Currently, AdAL has the following qualities:

The Screen

On a 24 by 80 screen there is a command line interface that can be used to enter one of the 19 commands or other data required by AdAL. There are five pop-down menus that allow the user to choose any of the 19 commands.

The Commands

F1 opens an Ada file and displays it on the screen 20 lines at a time. The user can scroll through the file by using the cursor keys and the NEXT SCREEN and PREV SCREEN keys. By moving the cursor to an identifier and pressing the RETURN or ENTER keys the user can choose to use that identifier as a name or keyword to find other Ada or LIL files or to create a keyword associated with that file. The default path name to Ada files is always (.ADA_CODE) and to LIL files is always (.LIL_CODE). These defaults can be overwritten so that searching and storing can be accomplished in other locations.

F2 is the same as F1 except that the file being viewed will be a LIL file.

F3 is the same as F1 except that any file can be viewed. The default path is the current path to ADAL.

F4 copies one or more files. This is essentially the same as the VMS "COPY" command.

F5 deletes a file. This is essentially the same as the VMS "DELETE" command.

F6 renames a file. This is essentially the same as the VMS "RENAME" command.

F7 quits AdAL.

B1 lists all Ada files in [.ADA_CODE] with a specified keyword associated with them.

B2 lists all LIL files in [.LIL_CODE] with a specified keyword associated with them.

B3 lists all LIL and Ada files with a common specified keyword.

C1 Creates keywords to a specified Ada file; the default path is [.ADA_CODE]. This "permanently" fixes all keywords created through an F1 command.

C2 is the same as C1 but for LIL files.

C3 Keywords created through F1, F2 and F3 are temporarily stored in a file that keeps a list of keywords chosen along with the associated file names. C3 will allow the user to edit this file.

C4 Fixes all common keyword relationships among Ada and LIL files.

(Not implemented: C5, C6 and C7 which will disassociate keywords that were fixed.)

B1 builds (edits) an Ada file. The default VMS editor is EVE (EDIT/TPU), and the default path is [.ADA_CODE]. These defaults can be altered. The edit commands are entered with unit names instead of file names along with a choice of type of Ada compilation unit: generic package, generic function, package, etc. New files are then created for editing with the basic structures already built into the buffer.

B2 builds a LIL file. The structure is approximately the same as B1. Unit names and a choice of type of LIL unit are used instead of file names.

B3 builds an instantiation of a generic unit in [.ADA_CODE] using the associations of the appropriate LIL and Ada files. The instantiation can then be placed in any directory for inclusion in an Ada unit.

H1 displays the keyboard help screen. The actions of special keys and combinations of keys are described.

H2 displays the commands described here and above.

(not implemented: O1, O2, O3, O4 which will pretty print, linking, compile, and list unit closures and dependencies.)

Dependencies

Of course this tool is very dependent on VAX/VMS routines and somewhat on the conventions of the VAX Ada compiler. We have plans to cut most of these dependencies and plan on implementing Adal on MS-DOS and UNIX based machines.

FUTURE EFFORTS

Besides the plans mentioned above there is still a large amount of work involved in inserting this tool in a practical environment. This will mean developing a more straight forward LIL syntax along with more automation of the processes. We have done some testing on the "standard" stack, queue, string, search, etc. packages with some success. We feel the real test will come in using Adal on a "reasonably large" project. Our future plans also include the writing of a LIL syntax checker and some sort of natural language approach to the searching process.⁴

1. J. A. Goguen, "Reusing and Interconnecting Software Components," IEEE Computer, Vol. 19, No. 2, pp.16-28, February 1986.
2. G. Gruman, "Early Reuse Practice Lives Up to its Promise," IEEE Software, Vol. 5, No. 6, pp.87-91, November 1988.
3. G. C. Harrison, "An Automated Method of Referencing Ada Reusable Code Using LIL," Proceedings of the Joint Ada Conference Fifth National Conference on Ada Technology and Washington Ada Symposium. pp.1-7, March 1987.
4. D.-B. Liu, "A Knowledge-Structure of a Reusing Software Component in LIL," Proceedings of the Sixth National Conference on Ada Technology. pp.337-380, March 1988.
5. W. Tracz, "Ada Reusability Efforts: A Survey of the State of the Practice," Proceedings of the Joint Ada Conference Fifth National Conference on Ada Technology and Washington Ada Symposium. pp35-44, March 1987.

About the Author

George C. Harrison is a Professor of Computer Science in the Department of Mathematics and Computer Science at Norfolk State University. He took his Ph.D. in mathematics from the University of Virginia in 1973 and his M.S. in computer science from Old Dominion University in 1986. His recent research interests include program verification and reusability theory.

Department of Mathematics
and Computer Science
Norfolk State University
2401 Corpview Avenue
Norfolk, VA 23504



REUSABLE SUBSYSTEMS FROM A HIGH PERFORMANCE ADA COMMUNICATION SYSTEM

Thomas L. Chen & Walter Sobkiw
ECI Division, E-Systems, Inc.
St. Petersburg, Florida

ABSTRACT

The reuse of functionally equivalent software is limited by performance and reliability requirements. The reusability can be improved when the software system is designed for each class of applications following requirements established for a reusable architecture. The reusable system is made up of functional objects and binding objects that follow a set of program paradigms. The functional objects and binding objects in a class of applications are mixed and recombined to achieve the best performance and reliability according to the hardware and operating system used to drive the application.

INTRODUCTION

The data communication industry, more than any other industry, is obsessed with standards and conventions. It can therefore be expected that there is a high degree of reuse of existing software in this industry. There is indeed a very high degree of reuse in this industry. This is evident by the popularity of SNA and DecNet. However, there is a continuous stream of communication software being developed from scratch. This is especially bothersome because most existing network software facilitates the installation of custom protocols where it is required.

The need for custom communication software is justified by the performance and reliability provided by existing software - on the hardware dictated by the application - which does not meet the requirement of the intended application. The reusability of existing software is then limited by the performance and reliability when it is installed on a given set of hardware. This paper presents an approach that manages reusability and portability for high performance data communication software.

CURRENT SOLUTIONS

There are two current solutions where software reuse has been successful. The first solution uses an existing data communication software system, augmented by custom protocols, that satisfies the performance and reliability requirements on the hardware and is acceptable to the functional applications.

The first solution is most desirable as long as the hardware required to deliver the required performance is not limited by:

- Space
- Cost
- Reliability
- Weight
- Power consumption
- Processing speed

The second solution uses existing subroutines or small software packages to support reusability for a new functional application. This solution offers very little saving because the majority of the software cost is in the design, integration, and system test. Most of the cost is not in coding and unit test.

Another possibility is the reuse of software subsystems from existing software systems to build high performance software for specific applications. Approaches for using software subsystems as opposed to whole software systems or small software units to support reusability have not been extensively studied. The advantages, disadvantages, and problems associated with this third approach are not well known.

CURRENT APPROACHES TO INCREASE SOFTWARE REUSE

The current design methodologies, whether they are structured or object oriented design, approach the software reuse issue on the following two principles:

1. Identify common functions through implementation independent functional decomposition or object identification.
2. Encapsulate the implementation of the common function, discussed by Cox and Hunt [1].

There has been a fair amount of success with this approach. However, 15 years after the introduction of structured system analysis, the software reuse problem is still a subject of significant study.

LESSONS FROM SUCCESSFUL REUSABLE SOFTWARE EFFORTS

There are many successful systems where parts of the system are rearranged, or augmented with new software, to form different applications. Most notable are:

1. UNIX PIPE for text file applications
2. Transaction processing systems like CICS

These reusable software systems have the following common characteristics:

- The software is only reusable in a Compatible class of applications. The different applications in the same class can be as diverse as an air cargo system or a bank debit system.
- A defined Binding Architecture that spells out the different subsystems comprising the system and defines how each piece should fit.
- Binding Objects that tie all pieces together but do not directly add to the functional requirements of the application.
- Functional objects that are directly related to the functional requirements of the application.
- All the functional objects and binding objects are constructed to a compatible Program Paradigm for the unique reusable system.
- Portable Language.

It is interesting to note that these successful reusable software methods were developed before structure programming, structure system analysis, and object oriented design were introduced into the software community.

Figure 1 shows how the hardware community has developed the capability to mix different subsystems into an appropriate system that can support multiple applications. This has been supported with the definition of standard hardware "Binding" objects that permit the linking of various functional objects to other functional objects. An analogy between the hardware and software community is shown in Tables I and II.

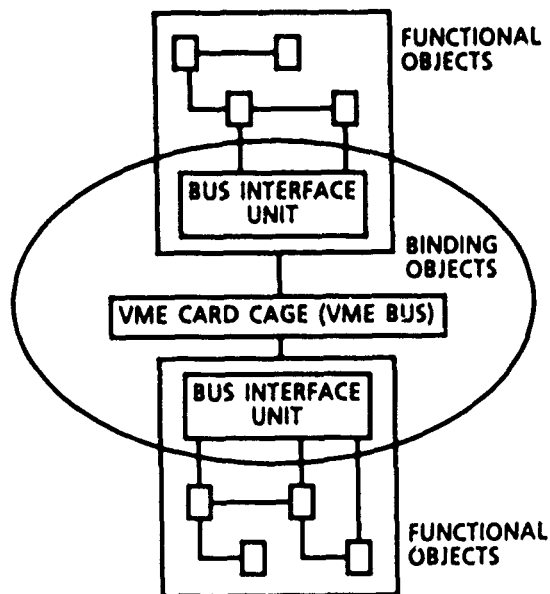


Figure 1. Hardware Reusability

Table I lists the hardware equivalents of the reusable system characteristics. Table II lists the different parts of the reusable software system according to the required characteristics.

Table I. Hardware Equivalent Partition Example

Class of Application	Binding Architecture	Binding Object	Functional Object	Compatible Paradigm	Portable Language
Non-based computer system	Non specification	Physical bus, pin layout, and bus interface units	Unique logic on application boards	VME multibus	Non portable hardware

Table II. Reusable Software System Partition

Class of Application	Binding Architecture	Binding Object	Functional Object	Compatible Paradigm	Portable Language
UNIX text file manipulation	Text file only Serial processing	Pipe I/O redirection	Group, cat, ls, type	Use standard input output device	"C"
On line transaction processing	Transaction processing program using guide	CKS program, transaction files and configuration files	Transaction programs	Six step program cycle, Atomic operation transaction driver	Unal Fortran M.I *Hal

*Not portable, reusable only

We can also draw the following observations from these successful software systems about software reuse:

- Easy rearrangement of existing functions to produce new applications is a key for software reuse.
- A narrowly defined class of application is often enough to support the additional cost of software reuse.
- Exact match of function is not necessary for reuse.
- Easy addition of new functions is essential.

REUSABLE HIGH PERFORMANCE COMMUNICATION SOFTWARE DESIGN APPROACH

The novel reusable software design approach described in this paper is based on observations of limited reusability in both the software and hardware community and on two considerations which are not advocated in current software engineering practice.

The first consideration is acknowledging that the total software solution includes extensive amounts of software executable code used to support the binding of the functional application software to each other, to the hardware, and to the operating system services. In this novel design process, there is a conscious effort to separate the purely functional pieces of software from all other software that is dependent on the hardware and operating system environment.

The second consideration is that the binding effort and the selection of the hardware is not a one time event in the life cycle of a software project. This is especially true in the high performance embedded system. This point was expanded upon in a discussion about fault tolerance and performance by Chen and Sobkiw [6]. Thus, if an effective mechanism could be developed to isolate unique "binding objects software" from "functional objects software," then not only will the potential for reusability increase, but also during the course of software development/modification, the effort may be reduced as functions are bound in different ways to support various stages of development. The functional design of the application and the elaboration of the binding effort, as well as the selection of the hardware, can be carried out as two independent activities if the two interfacing activities are properly defined.

Figure 2 shows that there is an area of software activity that eventually translates to unique code. That software effectively allows the application to become integrated with the operating system and hardware services. This is shown conceptually in Figure 3.

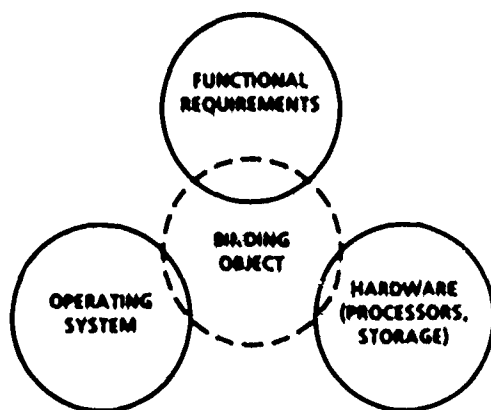


Figure 2. The Software Activity

A major piece of software is overlooked by today's design methodologies. This software binds the functional software, operational software, and hardware resources.

In structured system analysis the functional design is bound to the operating system and hardware after an implementation independent analysis. The same can be said of the OOD techniques in which the unique hardware architecture is bound with the resulting OOD based design. These design approaches were driven by two assumptions.

The first assumption was that the software design starts with an implementation independent analysis which defines functions and data flows. Then, the software functions are allocated to hardware resources. Each group of functions in a hardware resource can be allocated into software processes and these processes can be designated as a collection of procedures by structured system analysis or other techniques. This one time procedure is seldom successful. The allocation of processes in the data flow diagrams are either done according to the target system at the very beginning or are not used at all when the final software processes are allocated to the hardware. This practice is partly confirmed by Post [4]. Chen and Steimle [9] illustrate the drastic differences in the software design that performs the same application function but delivers different performance characteristics. A major portion of the software design is unaccounted for in the solution. The unaccounted for software in the design is the software that effectively links the hardware resources and operating system resources to the applications software.

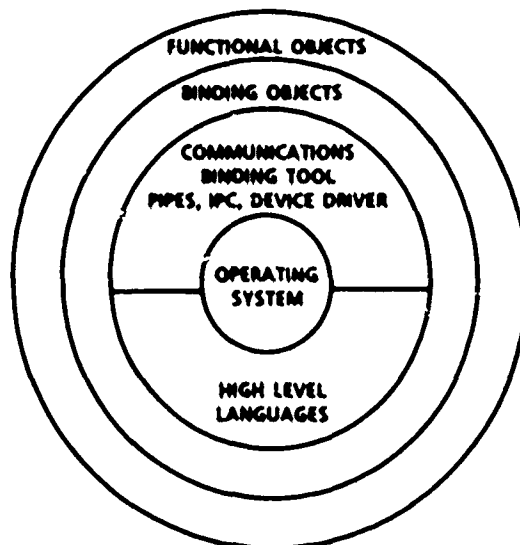


Figure 3. System Layers

The binding objects are not unlike the transaction support systems, provided by UNIVAC or ILLIAC, that address non-transaction oriented activities such as Fault Tolerant communications.

The second assumption was that the software designer does not need to understand the hardware being used in the system. In order to achieve performance, unique hardware and operating system control structures are used in the final solution. These structures control parallelism, manage storage, address data integrity and other key system characteristics. Karp [8] and Burger [5] elaborate on this point. This discussion on the explicit control of parallel activities and storage management can be defined as a binding effort to mate the application to the chosen hardware.

Figure 3 illustrates how the software in a system can be seen as a layered collection of elements. At the heart of this collection is the operating system which mates all the software to the hardware. Next come the languages, linkers, IPCs, and system configuration files that not only translate application program source code to executable code, but also define the profile of the application and bind the application to the services and facilities provided by the operating system. The application gains the services of the CPU and I/O by manipulating these services. The next layer is the binding objects layer. The outer layer of this collection of elements is the functional objects. The functional objects must follow the interface rules to the inner layer while satisfying the functional requirements of the application.

This picture is not new and there is an existing model for this concept in the form of transaction services. The transaction services of IBM, UNISYS, and other computer vendors allow multiple applications to be developed without recreating the software that links the primary mission applications software to the hardware and operating system services. This shell can be extremely large in terms of the total software effort depending on the system characteristics.

Examples include the transaction processing paradigm provided by Sperry TIPS 1100 and CICS supported by IBM. The binding objects are transaction processing support software items provided by the vendor. The transaction programs are discrete programs provided by the user that satisfy the functional requirements of the application. In the case of Sperry, these programs must be coded according to a style defined by TIPS 1100 and follow the interface rules to TIPS 1100. The same requirements are true for the CICS supported by IBM.

The issue is that if a software IC is to achieve reusability then that software IC should be purely functional in nature and not contain any "glue" to bind it to hardware or operating system services. In other words, the software IC should be separable from the architecture of each application. In addition, the success of a software "IC" is based on its firm, fixed, accepted interface definitions which effectively translate to the architecture of that software IC. The binding objects in Figure 3 must present a standard, well defined, well accepted interface to the functional objects.

The step taken to design this system is a pragmatic one. First, the architecture of the system is laid out to contain all the characteristics of a successful reusable system. Table III lists the architecture components of the reusable high performance communication system according to the required characteristics. Structured analysis, as well as object oriented programming technique, is used to build the functional objects and the binding objects as illustrated by Chen and Sutton [3].

Table III. Reusable High Performance Communication System

Class of Application	Binding Architecture	Binding Object	Functional Object	Compatible Paradigm	Portable Language
High performance software	UNIX processes, distributed hardware dedicated processor for asynchronous remote inputs	IPC, shared memory shared disk file Ada facade	Functional management entities	Single object for multiple asynchronous remote inputs interlocked network nodes	Ada

BINDING ARCHITECTURE

Within UNIX a high performance application with asynchronous inputs is made up of UNIX processes and device drivers. These UNIX processes and device drivers can be distributed into various hardware.

The UNIX processes are further divided into input processes and principle processes. Each input process is dedicated to an input. Sufficient input processes are created so that there is always a free input process available when the device driver receives an input from any device. The UNIX processes communicate to the device driver through file I/O. The UNIX processes synchronize with each other through shared memory, IPC, disk files, and interprocessor IPC. The UNIX processes are distributed across several loosely coupled processors.

Each UNIX process must be programmed according to a program paradigm which is compatible to the binding objects and the binding architecture. Each UNIX process in this application is made of the following components:

Binding objects:

- Main program
This is the software that ties all procedures together to form a UNIX process.
- System Access Packages (SAPs)
This is a procedure interface to procedures in a different process.

Functional objects:

These are the software packages that directly relate to some application functional requirements.

BINDING OBJECTS

Binding objects are those software items that tie all pieces of the application together but do not directly support application functional requirements.

- **Main program**

The main program is the one single part that ties all the packages together when the subsystem is used as a process. This program is individually developed for each process.

- **System Access Packages (SAPs)**

The system access point is defined as a software package which is independently developed to connect functional objects in different UNIX processes. Instead of interfacing directly, through IPC or shared memory, the functional objects interface with the system access point(s).

This concept is similar to the remote procedure call elaborated on by Wilber and Bacarisse [2]. A SAP contains three main parts:

1. Server interface package - A package specification.
2. Client interface package - A package specification.
3. Body objects - Several body objects are required for each SAP. There is one package body for each interface.

FUNCTIONAL OBJECTS

Functional objects are those software items that are directly related to the functional requirements of the application. The functional requirements of a data communication system can be partitioned into the following functions according to the ISO 7 layer model. The Data transportation functions are:

- Application
- Presentation
- Session
- Transport
- Network
- Link
- Physical

The Management functions are:

- Monitor and record
- Network management

Through standardization, each of component of this model is reusable in different applications. There are, however, many protocols. Therefore, each application that wants to incorporate reusable code must blend implementations of the protocols.

A high performance communication software system can be made up of the following two groups of UNIX processes:

Data transport:

- Front-end network process
- Back-end network process

Management entity:

- Monitoring and recording process
- Network management process

The Data transport subsystems have the following characteristics:

- They implement parts or all of the ISO 7 layer functions above level 2 protocol. The two lower level protocols are implemented as UNIX device drivers.
- They transport data from one connection to another connection. The sub-system usually consists of a protocol part and a set of tables describing each connection.
- They can easily be replicated and distributed over many computer processors.

The Management subsystems have the following characteristics:

- They communicate with all 7 layers of the ISO model.
- They either provide a centralized control for the whole communication system, or provide a centralized repository for the whole communication system.
- They can be distributed over many processors only in a hierarchical manner.

In the implementation of a customized communication system, these subsystems are combined, or split into a number of processes, and distributed on the selected hardware (computers) according to the operating system characteristics and the application traffic flow to obtain the best performance for the hardware chosen to host the communication system.

PORTABLE LANGUAGE

Ada is mandated by the contract for this data communication system. Its package features enable an elegant implementation of each SAP, shown in Figure 4. Ada package specification provides a nice Ada facade for each SAP that can be independently compiled. The features provided by Ada are severely hampered because each Ada linked output is implemented as a single UNIX process. The very large load module generated by Ada is also an issue of concern.

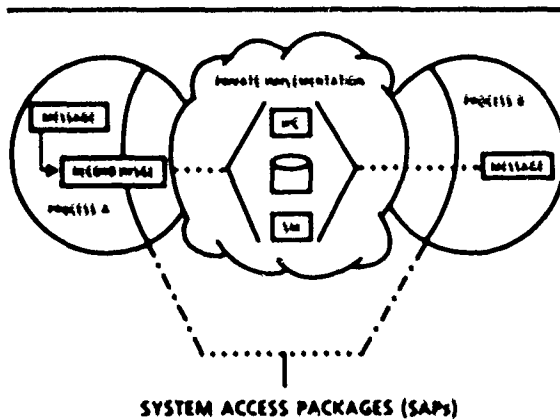


Figure 4. System Access Package (SAP)
The SAP interfaces the functional application to the operating system and hardware environment

PROGRAM PARADIGM

All the functional objects must be constructed according to compatible program paradigms for this class of reusable system. This programming style is developed in the traditional transaction processing system shown in Figure 5.

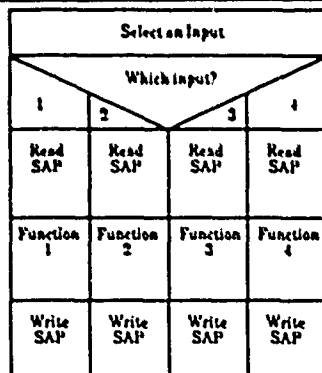


Figure 5. Compatible Program Paradigm Loop

Each transaction program is made up of one or more transactions. The transaction program accepts asynchronous input only in one predetermined location in the program. Each transaction is driven by one unique input. The transaction processes the input, updates related data base information, stores intermediate results or generates output, then loops back and waits for the next input. The programming style can be illustrated by the following example where a free style program is transformed to a transaction program:

1. Read A --asynchronous input
2. Read B --asynchronous input
3. C:=A+B Function requires two asynchronous inputs
4. Write C
5. Loop

A transaction program to accomplish the same requirement looks like the following. The sequence in each column is a transaction.

- | | |
|-------------------------------------------------------------------|----------------------------|
| 1. Read A or B -- Home position wait for inputs | |
| 2. If A Else | |
| 3. If B is on queue 3. If A is on the queue | |
| then then | |
| get B -- Synchronous input | get A -- Synchronous input |
| C:=A+B | C:=A+B |
| write C | write C |
| Else | Else |
| put A on queue | put B on queue |
| exit or loop | exit or loop |

Synchronous inputs are inputs that the transaction can get on demand. These inputs are stored in shared memory or in local disk.

CONCLUSION

Our attempt to build a high performance communication system incorporating reusability was marginally successful. We demonstrated that when major functions are rearranged, a specific level of performance can be achieved. We showed that Functional objects are reusable and portable; that Binding objects are reusable but not easily portable to different operating systems; and the use of an enumeration type makes the addition of a new function difficult.

The ECI approach to reusability was attempted on a high performance comm system with approximately 100K lines of Ada code. Although the program did not specifically identify reusability requirements, we did include an effort to identify characteristics which supported reusability.

The design required several physical allocation changes in the development cycle to achieve the best performance and reliability goals. These changes were accomplished with no changes in Functional objects. This provided confidence that the function aspects could be ported to different system hardware platforms to combine with new Binding objects to carry out the same application. This approach to reusability accommodates the different architecture requirements to achieve the best performance and reliability in difficult hardware platforms.

In summary, this novel approach to reusability is based on acknowledging that systems consist of functional and architecture dependent code. Given this assumption, the design process includes separating functional code from architecture code early in the effort and defining a binding mechanism that uses existing services, the SAP and the mainline.

We recognize that our paradigm needs extensive refinement and expansion to provide the level of reusability needed in current Ada applications. ECI expects to continue its research into the applications of reusability in three existing programs, and will attempt to expand its present data base to other systems through several mechanisms now under active research.

REFERENCES

1. Brad Cox and Bill Hunt, "Objects, Icons, And Software-ICS", Byte, August 1986, pg 161.
2. Steve Wilber and Ben Bacarisse, "Building Distributed System With Remote Procedure Call", Software Engineering Journal, September 1987, pg 148.

3. T. L. Chen and M. Sutton, "Object Oriented Design: Is It Enough For Large Ada System", Proceedings of 1988 ACM Computer Science Conference, pg 529-534.

4. J. Post, "Application Of A Structured Methodology To Real Time Industrial Software Development", Software Engineering Journal, November 1986, pg 222-234.

5. A. H. Karp, "Programming For Parallelism", Computer, Vol. 20, No. 5, May 1987, pg 43-55.

6. W. Sobkiw and T. L. Chen, "Design For Fault Tolerance And Performance In A DOI-STD-2167 Ada Project", Proceedings of the Sixth National Conference on Ada Technology, pg 424.

7. R. P. Wiley, "A Parallel Architecture Comes Of Age At Last", Spectrum, Vol. 24, No. 6, June 1987, pg 46-50.

8. T. M. Burger and K. W. Nelson, "An Assessment of The Overhead Associated With Tasking Facilities and Task Paradigms In Ada", SigAda, Vol. VII, No. 1, pg 48.

9. T. L. Chen and C. L. Steimle, "Two Design Approaches Using The Ada Language", IEEE Southeastcon 87, Vol. 1, pg 72.

BIOGRAPHY

Thomas L.C. Chen is a member of the Technical Staff in the Software Systems Department, E-Systems, ECI Division. He is the principle software designer of Survivable Communications Systems, has over 25 years experience in the development of communications methodology. He holds a M.E. from Taipei Institute of Technology.

Walter Sobkiw is a senior principal engineer with E-Systems, ECI Division. He is currently a member of the Advanced Technology Team and is responsible for defining system development methodologies and new business pursuits. He holds a BSEE from Drexel University.



Constructing Domain-Specific Ada Reuse Libraries

James J. Solderitsch, Kurt C. Wallnau, John A. Thalhamer

*Unisys Defense Systems
Paoli Research Center
PO Box 517
Paoli, PA 19301-0517*

Abstract

High-Impact reuse is achieved by focusing on specific Application Domains. A Software Component Reuse Library System must support domain modeling as well as repository management features. The RLF project addresses both of these areas. Repository management capabilities including retrieval, classification, insertion and qualification of components are all provided. Domain modeling is achieved through knowledge representation components that were developed in Ada using an Ada perspective. The domain model provides an effective and powerful interface to the library. An evolutionary approach has enabled the production of a family of librarian applications of varying functionality and point-of-view. Ada features, such as generics and exception handling, and Ada design principles, such as data abstraction, are used to construct systems that incorporate traditional AI functionality while providing enhanced system maintainability and evolvability.

1. Introduction.

This paper describes the accomplishments and experiences of the Reusability Library Framework (RLF) project* being performed at the Paoli Research Center. The RLF project, as part of the STARS (Software Technology for Adaptable and Reliable Systems) Foundations program, was proposed, designed and is being implemented to meet several goals.

Our primary focus is the development of framework technology to support the creation of software reuse libraries (or repositories) that provide for an intelligently-guided search through a library of software components, and more generally, a knowledge-based approach to the management of software artifacts that apply to a particular application domain. To achieve this primary goal, we developed reusable, stand-alone Ada subsystems to provide the necessary knowledge-based technology. These components form the underlying structure upon which librarian applications are constructed. As a demonstration

that we have produced the required tools and capabilities, we are creating a proof-of-concept Ada component library hosted on the framework.

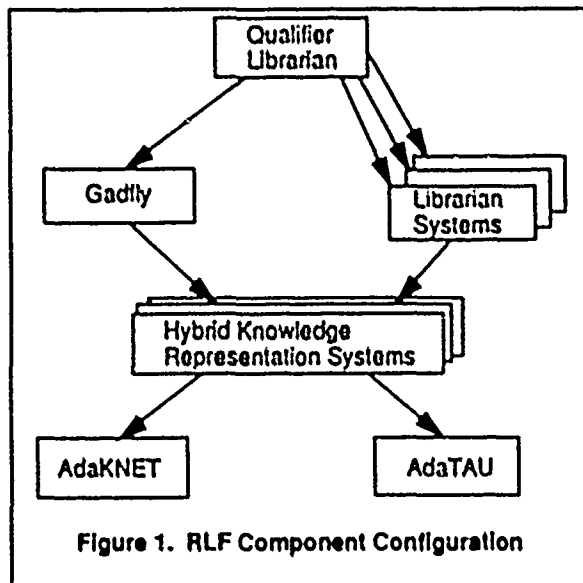
Our approach to meeting these goals was first to examine alternate approaches to knowledge representation in the literature as well as in Artificial Intelligence (AI) projects conducted at Unisys. We identified two complementary technologies from this search and developed Ada designs and implementations that provide a powerful technological base on which to host librarian applications. AdaKNET (Ada Knowledge NETwork) and AdaTAU (where TAU is an acronym denoting the phrase Think—Ask—Update) resulted from this analysis.

AdaKNET enables the creation of structured inheritance networks (sometimes called semantic networks) which are used to model the structural component taxonomies that evolve whenever long-lived and large scale software development takes place in particular application domains. AdaTAU provides a rule-base formalism used to conduct inferencing over particular knowledge networks. In particular, AdaTAU provides expert assistance for the novice library user in browsing, searching and installing software components in the repository.

Both of these components may be integrated into hybrid knowledge representation systems and to demonstrate this we developed a knowledge-based Ada unit test assistant called Gadfly. This system uses a coordinated knowledge base about Ada units and testing methodologies to generate unit test plans. These plans are based on a parsed representation of individual Ada components and a rule-base-guided dialogue with the user. Such a dialogue is designed to probe for relevant assumptions regarding the construction and intended use of the unit. Gadfly's design allows it to function as either a stand-alone test plan assistant, or as a testing/quality assurance module as part of a complete librarian application.

The interconnection of these components and their integration into librarian systems of varying strength and functionality is illustrated in Figure 1. At the top of this diagram, the qualifying librarian denotes our final delivered system which includes Gadfly as an important module used to validate library components that are offered for

* The work described herein was funded by STARS Foundations contract number N00014-88-C-2052, administered by the Naval Research Laboratory.



inclusion in the library. This system should be viewed as a demonstrable prototype of the representational and managerial power that we provide, and the integrability of the subsystems that we have developed. Leading up to this inclusive prototype, we have developed *Intelligent* librarian systems of smaller size and functionality aimed at exploiting various aspects of our underlying technologies. These systems all support the basic search, retrieval and insertion of software components in library settings based on domain-specific knowledge stored in knowledge bases encoded by AdaKNET and AdaTAU.

Section 2 presents a discussion of the power and usefulness of constructing libraries based on a domain model with our prototype domain of Ada benchmarks providing illustrations. Major components of the project shown in Figure 1 are described more fully in sections 3 and 4 of this paper. The knowledge base components and Gadfly are themselves the subject of a paper delivered at the AIDA - 88 conference [Wallis] and the interested reader is encouraged to consult this paper to expand upon the relatively high level view of the knowledge base components and Gadfly that is given here. We close the paper with a section describing some of our Ada experiences and lessons-learned and a final section that summarizes project results and presents some of the conclusions that we have reached as a result of working on the project.

2. Domain-Specific Libraries – An Example

A major goal of Ada developers is to reduce code development time by the use of reusable code. One difficulty in code reuse has always been locating and selecting appropriate existing code. Another has been in making use of code which may have been written for a quite different purpose, with different assumptions and needs. Modifications must be made with care, without inadvert-

ently violating some assumptions and therefore introducing errors. The purpose of the Reusability Library Framework is to provide the user with intelligent assistance in inserting, retrieving and using Ada code modules located in a software parts library.

In order for reuse technology to succeed and permit the large scale productivity increases that were forsook back in the early days of software engineering [McIntire], such technology must go beyond the basic notions of subroutine libraries, directory-based repositories and simple software catalogs. A domain model approach to reuse libraries has the potential to provide for significant productivity boosts. This assertion is supported by the following evidence.

Collections of modules that have been extensively reused such as the well-known Fortran mathematics and statistics libraries were organized around particular, clearly understood domains. Moreover the reusability of a particular component is itself a relative property of its place within an application domain. The design of a reusable component has been compared to a kind of market analysis to determine important properties and features of a component, along with performance requirements [McCabe]. A repository must provide explicit support to application developers working in specific application areas. Each such applications area will require a specialized set of capabilities that are themselves highly reusable [Betz]. Domain-specificity also enlarges the granularity of reuse, supporting system composition in terms of subsystems tailored according to the requirements of new or modified applications.

The representational capabilities required of a domain-oriented reuse system go beyond those provided by a project database, but are less than those required of a complete CASE (Computer-Aided Software Engineering) environment. Future CASE environments must be constructed to support large scale reuse efforts and so domain modelling must be integrated into future CASE systems. Thus a reuse library system must support system development and not merely component retrievals. Researchers have already pointed out the importance of domain-specific knowledge in the software development process [Barua]. A modern library system must also support alternative technologies for system development; for example, a constructive approach through component composition and a generative approach through component specifications [Simo].

To date, there is no widely accepted "Dewey-decimal" classification system for software libraries. There are certainly no standard hierarchical systems, and no standard indexing schemes supporting retrieval from such storage hierarchies. The RLF approach supports the experimentation with many different classification schemes, each tailored according to particular application domains. The RLF also supports tailored search and retrieval ca-

pabilities depending on the domain as well as a user's skill level and personal preferences. By contrast, other published approaches to library management are fundamentally weaker.

Other projects have chosen to focus on either a hierarchical approach or a data base approach. Hierarchies provide a rigid classification scheme which enable component retrievals based on searches of the hierarchy. The database approach requires the user to construct database queries which may or may not produce a component of the kind being sought. Recent projects have begun to experiment with limited hybrid systems that combine the two approaches [Pri87] [Sunt87]. In any case, these approaches are limited by their support for a single, fixed taxonomic model, and their presentation of a single user interface supporting only one category of library user.

The RLF framework provides support for both dynamic and experimental classification systems as well as multiple classes of users. By using knowledge-base components, the RLF is able to support a separate, declarative domain model with powerful and adaptive taxonomic power along with tailorable search, retrieval and compositional capabilities. The domain model exists independently of the repository of components and expands upon the services provided by competing library organizations.

The RLF framework enables the definition and use of taxonomies for classifying Ada code and code-related information. This framework uses AdaKNET to provide a structure into which specific Ada components can be inserted. The framework supports multiple orthogonal taxonomies; thus code can be classified by function, by structure, and by any other relevant characteristics. The current project will create a subset of a complete library instance, concentrating on the domain of Ada benchmarks.

In addition to the generic framework, the Library must have content. In order to demonstrate the feasibility of the framework, the RLF project will populate it with benchmark routines drawn from several sources. These sources include the Ada Compiler Evaluation Capability (ACEC) suite developed by Softech, the Performance Issues Working Group (PIWG) suite developed by the ACM Special Interest Group on Ada (SIGAda), and the Hughes Aircraft Company Ada Benchmark Suite.

Organizing code modules into appropriate taxonomies facilitates their retrieval and use. However, reuse is aided substantially if the user is also provided less formal information about the domain or the code. This kind of information can include guidance about the most appropriate code, indications of other needed modules, etc. It represents the kinds of *rules of thumb* that experts in a particular area develop, to provide shortcuts and improve efficiency. This compilation of expert knowledge will be

represented in the Reusability Library Framework as AdaTAU rules.

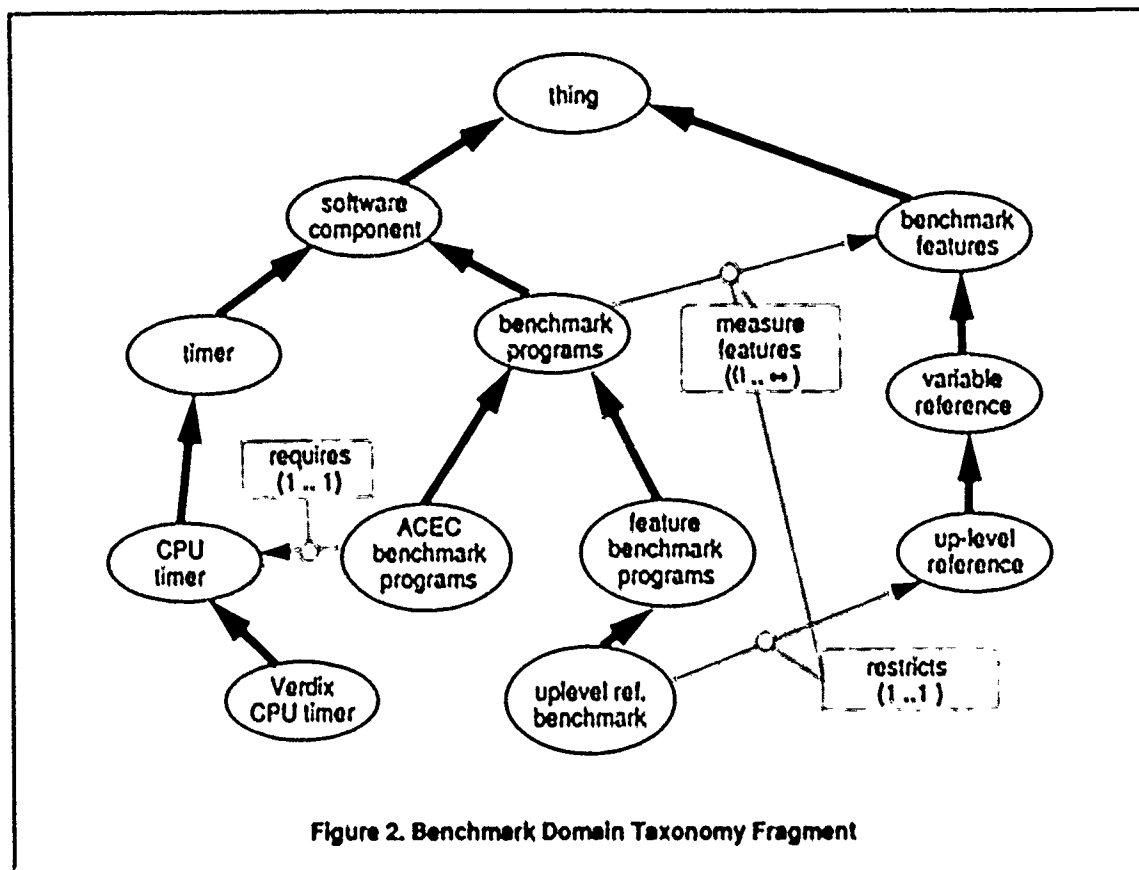
Figure 2 illustrates a small fragment of the Ada benchmark taxonomy that is being developed as part of this project. This fragment models the architectural decomposition of benchmark programs as well as the set of Ada features that are of interest to benchmark creators. An individual benchmark component is categorized according to this classifying framework. Such an individual inherits properties (attributes) from all of the categories of which it is a member.

The figure shows that the code category is broken down into timer and benchmark program subcategories. Benchmark programs measure Ada features. This relationship is indicated by the labeled arrow between the two categories. Any number of features can be measured by a benchmark and so the benchmark program category shows an unlimited range for the number of measured features. The figure also shows that benchmarks programs are broken down along suite membership as well as feature measurement. In particular, an ACEC feature benchmark must be equipped with a particular timer module in order that the benchmark be executed properly. Furthermore, timer programs are themselves delineated according to the Ada run-time system being utilized.

As can be seen, the web of relationships and dependencies can become complex even in a narrow portion of a narrow domain. The RLF framework not only supports the capturing of this information, but through integrated rule bases it permits an expert's view of the taxonomy be given to even novice users. For example, the user can be aided in browsing the domain model as well as in configuring an application (in this case assembling meaningful benchmark programs). Exactly the right timer is retrieved that matches the selected ACEC feature benchmark which measures the desired feature within the desired Ada run-time platform.

3. Knowledge Representation Systems

Underlying knowledge representation systems are critical to the building of domain-specific libraries. Thus, such systems are the underpinnings of the RLF project. AdaKNET and AdaTAU are derived from Unisys systems that were developed in Prolog and which have successively been applied to various problem domains. These systems were themselves designed from well-accepted knowledge representation and processing methodologies as developed in the AI literature. However, our approach was not to capture the style or approach of traditional AI systems implemented in Lisp or Prolog, or to create large Ada systems that emulate significant portions of Lisp and/or Prolog, but rather to analyze the objects and operations provided by proven AI subsystems and to rede-

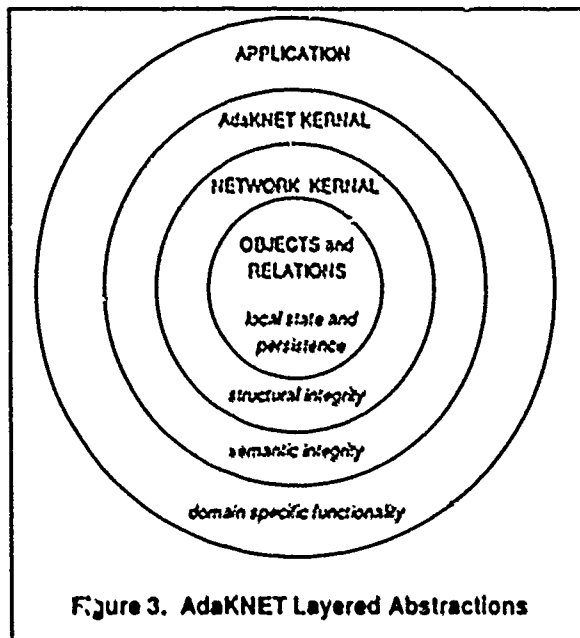


sign (engineer) and implement them from an Ada perspective. In particular, we examined candidate systems with an eye toward exploiting the strengths of the Ada language when achieving actual implementations. We tried to view various knowledge representation schemes in terms of the services they provide and the underlying structures used to enable these services. In short, we took an abstract data type point-of-view.

This approach naturally led to a convenient and important separation of the procedural and declarative knowledge that together permit knowledge-based processing to occur. The content of domain-specific knowledge bases is defined through the use of specification languages. Knowledge-base processing occurs via the execution of operations defined in the knowledge-base components. Furthermore, using an Ada perspective led to a layered implementation plan that resulted in increasing levels of functionality and semantic checking being built upon lower, and more primitive layers. The following subsections briefly discuss the individual subsystems and their hybridization. Gadgetry is discussed as one particular example of hybridization. We also illustrate the specification languages with a couple of examples.

AdaKNET

Figure 3 shows a skeletal description of the design of the AdaKNET system. AdaKNET is based on a proprietary system developed at the Paoli Research Center called KNET [Free83]. KNET is itself derived from a well-known knowledge representation system called KL-ONE [Brace1]. KNET has been successfully applied to the area of (computer) system configuration. AdaKNET (and KNET) supports the management of two basic relationships between objects and classes of objects: *specialization* and *aggregation*. Specialization captures the sort of knowledge that one thing, or category of things, is a special case (or kind) of another thing. For example, the category of Ada subprograms includes both procedures and functions, while the category of Ada compilation units includes both packages and subprograms. Aggregation captures the fact that one thing or category of things is partially defined by its constituents or its properties (sometimes called attributes). For example, an Ada package specification is defined by its name, the types it exports, and the operations it exports. Also, an Ada software system is defined in terms of its subsystems which are defined by the constituent packages of the subsystems.



AdaKNET's representational power comes from its modeling of these two relationships, and their coordination through the use of inheritance. Inheritance is the capability that a subcategory of a category automatically inherits all of the attributes of the parent category (called super categories). The attributes of a subcategory may refine those of a parent category, but may never contradict them. However, a subcategory can admit entirely new attributes not defined for the parent. This inheritance is not local (one level only) but completely transitive where attributes of far-away parents are available at any node. AdaKNET rigorously enforces the sort of specialization semantics required by such a system, where violations give rise to Ada exceptions.

We have experimented with various forms of inheritance and recently have implemented a form of multiple inheritance where a subcategory of two or more super categories is equipped with the attributes of all super categories. Multiple inheritance permits smaller and clearer specifications of relationships that commonly occur in a software component library. For example, a particular benchmark program can simultaneously be a kind of feature benchmark as well as a member of a particular benchmark suite. Through multiple inheritance, each of these benchmark categories bestows a set of attributes that must be filled by particular values. The knowledge representation model provided by AdaKNET is a natural extension of a library framework taxonomic model where subject areas are broken down through extensive specializations, and where sub-categories within a specialization are distinguished by different feature sets (attributes).

By modeling information independently from the applications which are written to make use of this knowledge,

the information can be reused in multiple applications. For example, the Ada unit model defined to serve the Gadget application is usable as is within any of the library applications. The information itself is available in ASCII text files of SNDL (Semantic Network Description Language) specifications. A SNDL example describing a small part of the Ada unit model is shown in Figure 4. Some background on the use of specification languages within AdaKNET and AdaTAU is given later in this section.

AdaTAU

Like AdaKNET, AdaTAU is based on a Unisys-proprietary system written in Prolog called TAU (for Think, Act, Update). TAU was initially developed as a stand-alone rule-based system supporting diagnosis and maintenance of faulty computer equipment. In its latest form, TAU has been incorporated into the KSTAMP system (Matusz) where it functions as a diagnostic assistant for the repair of Postal Service mail processing equipment. KSTAMP is also an example of a hybridized system that combines two different knowledge representation systems (KNET and TAU).

AdaTAU processes two kinds of information: rules and facts. A rule is a statement that if certain facts are "true", then other facts should be established as true. A simple rule can be expressed as:

```
if (parameter_1 is type integer) then
    (test_case_set_type is
        integer_test_case_set).
```

This example shows a single necessary fact (called an antecedent) and a single resulting fact (called a consequent). In AdaTAU, there may be multiple antecedents and consequents. A more complex kind of rule asserts particular consequent facts depending on an answer given by the user to a posed question.

Facts are simple statements which are collected into fact bases. All established facts are placed into a fact base which is accessible for the purpose of checking the applicability of rules (and therefore the addition of new facts). The state of an AdaTAU inference session is described by the current collection of rules as well as the current collection of known facts. In its simplest form, AdaTAU processes an initial collection of rules, along with an initial collection of facts (perhaps empty) to the point where no new facts can be added because no rules are left whose antecedents are in the fact base. Typically, after a rule is applied (or "fired"), it is marked so that it is no longer considered for firing. The end result of an AdaTAU execution is the final collection of facts which an application that invokes AdaTAU can process in an application-specific way.

```

concept Software_Component (Thing) is
  local roles
    Component_Name (1..1) of Text;
    Description (1..1) of Software_Component_Description;
    Location (1..1) of Component_Location;
    Auxiliary_required_components (0..infinity) of
      Software_Component;
    Subunits (0..infinity) of Software_Subcomponent;
  end local;
end concept;

concept Ada_Compiler_Benchmark (Software_Component) is
  local roles
    Measured_Features (1..1) of Ada_Features;
  end local;
end concept;

concept ACEC_Benchmark (Ada_Compiler_Benchmark) is
  local roles
    Control_Measurement_Component (1..1) of Software_Component;
    Instrument_Package (1..1) of Software_Component;
    -- This package is used for timing the control and the
    -- feature.
  end local;
end concept;

```

Figure 4. SSDL Fragment

Figure 5 describes layers within AdaTAU, including the provision of a "distributed" version of AdaTAU based upon a simpler centralized scheme (indicated by the inner two layers). Currently the centralized form of AdaTAU supports two rule types. IRules (Inference Rules) fire automatically when their antecedents are determined to be present in the fact base. QRules (Question Rules) are rules which do not directly add facts to the fact base but rather schedule the posing of questions to the user. The antecedents of a QRule must be true before the question is scheduled. The answer given by the user determines which collection of facts is added.

The distributed form of AdaTAU supports the use of multiple collections of rules, collected into inference bases, and both local fact bases (to record information established during a local inference session) and global fact bases (to record information that is transferable from one inference session to another). FRules (Focus Rules) are provided to support distributed AdaTAU. FRules enable the suspension of one local inference session in favor of another one. Distributed AdaTAU allows the partitioning of inferencing capability into localized distributed inference bases. An application-specific partitioning scheme supports cooperative focusing to the most appropriate inference base (where appropriateness can be judged from the contents of the global and local fact bases).

The actual expression of rules and facts is accomplished by a RBDL (Rule Base Definition Language) specification

file. A RBDL fragment is given in Figure 6. In addition, a RBDL specification file must declare a fact base schema definition to restrict the sorts of facts that can be installed into a fact base. In this way, AdaTAU provides a typing mechanism on facts analogous to Ada's own typing mechanism.

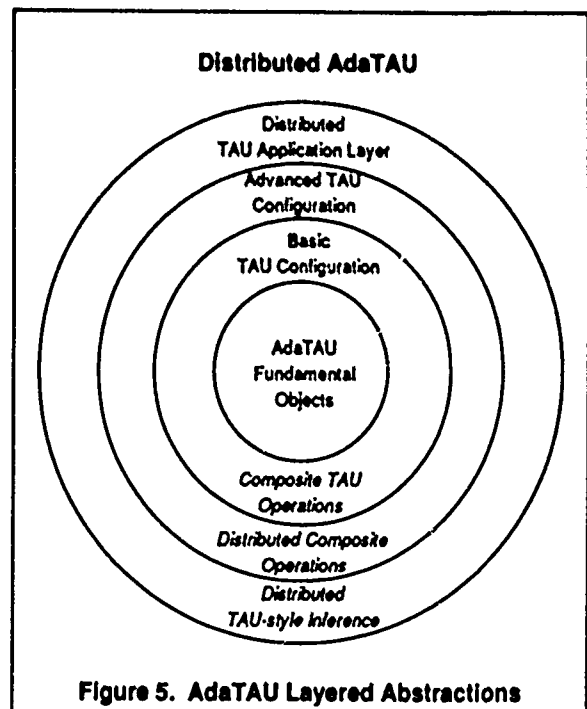


Figure 5. AdaTAU Layered Abstractions

```

fact base schema Ada_Compiler_Benchmark_Facts is
    access_mode : one_of (retrieve, insert);
    benchmark_purpose : one_of (feature, composite, suite, now);
    suite_choice : one_of (ACEC, PING, help);
    suite_member : one_of (ACEC, PING, help, unknown, none);

end Ada_Compiler_Benchmark_Facts;

question Ask_Benchmark_Purpose is
    text : (What purpose do you have in mind for the benchmarks?);
    type : one_of;
    responses :
        "test specific features of a language" =>
            (benchmark_purpose, feature);
        "test a mix of features of a language" =>
            (benchmark_purpose, composite);
        "choose a compiler based on overall performance" =>
            (benchmark_purpose, suite);

end question;

grule Determine_Benchmark_Purpose is
    antecedent : (access_mode, retrieve);
    question : Ask_Benchmark_Purpose;
    weight : 1;
    justification : (Help the user choose the next level according to
                    his needs);

end grule;

```

Figure 6. RBDL Fragment

Forming Hybrid Systems – Gadfly

In designing AdaKNET and AdaTAU, we were convinced that although these systems are useful separately, they would be most useful within the software reuse library framework when they are combined and integrated with each other. We discovered that the forming of such hybrid systems is not an easy task. One of the issues we faced is the degree of coupling that should be imposed on the two systems—whether the subsystems should be tightly or loosely coupled with one another. Another is the degree to which application-specific information can be migrated to the knowledge bases maintained by AdaKNET and AdaTAU, and kept out of the actual Ada code that provides the application.

Our work on Gadfly, the Ada unit test assistant, resulted in a tightly coupled arrangement. Certain nodes in the semantic network are equipped with inference bases as provided by AdaTAU, including a local fact base. The Gadfly application itself controls how these inference bases are consulted and provides a communication mechanism so that facts can be transferred into and out of local fact bases.

Gadfly provides a stand-alone Ada unit test assistant. Briefly, through SNDL specifications, a generic Ada unit

model is defined for Ada subprograms, and a subset of Ada type semantics is included for the model (for example, in our demonstrable prototype of Gadfly, semantics covering integer and file parameters are included). Another part of the Gadfly/SNDL specification includes a testing model to break down testing strategies based on Ada types and testing methodologies. A third form of knowledge is recorded in multiple Gadfly/RBDL specification files. These files contain test case heuristics drawn from "common sense" testing, general Ada semantics and particular properties of individual data types.

In our Gadfly prototype, we provide for a black box testing strategy broken down along the parameter modes of Ada, as well as the supported Ada data types. We complement the SNDL portion of the knowledge base with RBDL specifications of the sorts of questions to ask the user regarding design and usage assumptions about the unit under test. These rule bases are located at testing category nodes within the SNDL-specified semantic network.

When Gadfly is executed, the source code for an Ada specification is parsed to yield an instantiated form of the Ada unit model. Gadfly then automatically walks the user over the semantic network corresponding to this individual Ada unit and generates particular facts pertinent to the

testing of this Ada unit. The test-related facts deduced by Gadfly are based on the responses the user gives to the questions being posed as well as the particular testing model contained within Gadfly. Finally, another network walk is conducted during which stored local facts are converted to test case recommendations.

The integration of AdaKNET and AdaTAU as accomplished in Gadfly was our first experience in performing this hybridization and our approach to integration remains very experimental. For example, Gadfly was completed before the implementation of distributed AdaTAU was itself complete. In particular, Gadfly does not make use of focus rules to control transfers from inference base to inference base, but rather accomplishes such transfers within the Gadfly application itself. The librarian applications take advantage of the carefully arranged focus switches provided in distributed AdaTAU. However, like Gadfly, particular inference bases are located at particular nodes within the semantic network.

Knowledge-Base Description Languages

Our approach to defining knowledge-base processing components was to view them as abstract data types, and layers of such types. In defining these packages we concentrated on the necessary operations to support applications that required knowledge-based processing, and provided the underlying objects and object management to support such client applications.

The RLF project was conceived and executed in the context of available technology that supports the creation and use of special purpose high level languages. Such languages permit the needs of specific application domains to be expressed in terminology appropriate to such domains. In our case, we needed a way to specify the contents of knowledge bases in language that was clear and concise. Furthermore, such descriptions must be processed into a form that leads to installation into the data structures managed by AdaKNET and AdaTAU. The abstract data type (ADT) approach to the design of knowledge representation systems led to a large collection of ADTs with complex interactions. Knowledge-Base specification languages hide the procedural complexity of the ADTs and permit easy-to-understand instantiation of knowledge bases.

The use of such specification languages hinges on the availability of support for their design and implementation. Special purpose, "little" languages are a nice idea, but unless they are economical to develop and apply, they most probably will not be adopted. At Unisys, a tool exists that not only supports language definition but the tool is written in Ada and generates Ada code to accomplish specification translations. SSAGS [Pay82] (Syntax and Semantics Analysis and Generation System) has been in use since 1982 (fully operational in an almost total Ada configuration since 1986) and has been applied

in the development of a number of special purpose languages [Pom87].

Using SSAGS, we developed syntax for both forms of knowledge representation required for the RLF and corresponding semantics for expressions written in these languages. SNDL and RBDL are different because they support different sorts of information. However the translation principles that lead to the generation of Ada code from these languages are the same for both languages. From a SNDL/RBDL "spec", the SNDL/RBDL translator generates Ada code to initialize the persistent (currently Ada file based) forms of the data contained in the specifications themselves. To actually initialize the machine readable data structures, the generated programs need merely be compiled and run.

AdaKNET and AdaTAU themselves contain reverse translators that enable the current state of individual or hybrid knowledge bases to be saved as ASCII text files of SNDL and RBDL specifications. Thus, if any changes to knowledge bases are made interactively, a snapshot of these knowledge bases can be taken. SNDL and RBDL specifications are themselves highly portable and are our preferred means of porting knowledge bases to new computing environments.

Future Extensions

There are many possible extensions to make to AdaKNET. One feature of KNET, the Prolog progenitor of AdaKNET, that is currently not provided in AdaKNET is the notion of constraints whereby attributes of a subcategory can be automatically delimited when subcategories, or individuals of the subcategory, are created. Without some automatic way of performing such limitations (or constraining actions) the AdaKNET user is forced to do them by hand. More generally, experimentation with providing interfaces to AdaKNET that support the integration of AdaKNET with other knowledge representation systems would be desirable.

AdaTAU is particularly well-suited to providing "expert" navigational guidance through a complex domain model such as those definable with AdaKNET. Because the rule abstractions are simple (and AdaTAU rule bases are implemented via Ada generics), modifications to AdaTAU to support new kinds of rules or inference strategies are relatively easy to accomplish. We have already experimented with different sorts of partitioning and focusing schemes in building Gadfly and librarian applications. Other possible extensions to AdaTAU include supporting non-monotonic reasoning where rules can identify facts to be removed from the fact base as well as added to the fact base. Careful truth maintenance must be provided so that fact bases remain consistent, and rules can be "unfired" due to the removal of antecedent facts. As is the case for AdaKNET, it is possible to experiment with different kinds of application interfaces to AdaTAU

whereby application-specific actions can be invoked under the control of AdaTAU. The results returned from these actions can then be fed into the general inference process.

We plan to continue with our hybridization experiments. To support this research, an Integrated Knowledge-Base Description Language (KBDL) is planned which will support proven hybridization schemes as they are developed. We also intend to continue our development of Gadtly, both in support of librarian applications as well as a separate application. Our primary methodology will be to concentrate on knowledge engineering extensions. Planned knowledge base enhancements include support for additional Ada units such as packages as well as other Ada data types. Improved and enlarged RBDL specifications to support more sophisticated testing methodologies and parameter relationships (such as aliasing) are also under consideration.

4. Librarian Applications

The Reusability Library Framework project is focused on producing a whole family of librarian applications culminating in a qualifier librarian that includes Gadtly as a testing agent/quality assurance module, and which supports full repository management services over a software library focused on a particular application domain. There is nothing inherent in the approach taken in constructing the RLF that precludes it from being applied over a larger, less focused collection of software components. Although a general purpose reuse library could be built based on the RLF components with very good results, we believe that the benefits realized from the reuse of software components will be greatest in the near term, where software repositories are organized around narrow domains.

Not only will an RLF-based repository store individual modules and contain knowledge about their construction and quality, it will also possess knowledge concerning how components from the library have been successfully combined into subsystems that solve particular problems within the application domain. Knowledge about failures of components when used in inappropriate contexts can also be maintained. In short, a domain-focused library should contain all necessary information so that new personnel coming to work in the application domain have access to the collective wisdom of past system designers and implementors.

New system construction within such a library-based environment can be focused on the use of components found within the repository, with new module design and construction being pursued only as a last resort. When new parts are constructed, they should be designed with a clear understanding of their storage and reuse in a repository setting. Relationships to currently installed modules must be established when the new part is checked into the system. In addition, a quality analysis session

should be required where the offeror of a new component is queried regarding explicit and implicit assumptions about the design, testing and intended usage of the component. Such a dialogue session will ensure that a prospective user of the component has sufficient information to choose among several candidate components for a particular application.

Browsing Librarian

Our first sample librarian application is simply called `Library_Browser`. Its design was adapted from the design of an early browser-editor application which was used to test the functioning of AdaKNET. `Library_Browser` relies on a user-guided approach to navigating within the library (SNDL-provided) knowledge base. This method of user-controlled navigation might be the preferred method in a small library or for a user who has consulted the library frequently (or a domain expert who participated in its construction).

From a particular library category, the user can navigate to any other category or subcategory by typing its name. The attributes of a particular category can be examined as a group or individually. A history list of all recently examined categories is provided so that the user has a sense of where he or she has been browsing in the library. It is possible to generate a list of all subcategories of a given category so that a sense of the depth and breadth of the classification scheme can be obtained.

Once the user has navigated to a location where particular components have been identified, the file system location of these software components can be obtained along with particular values of attributes that the software component must have based on its membership within a particular component category. In early librarian versions, the actual location of the component is provided, rather than the component itself. The component can then be inspected with a system editor.

This first version also provides for insertion and deletion of both knowledge and software components by the expert user. Such privileges were naturally present in the AdaKNET browser-editor where all aspects of managing the semantic network were under scrutiny. In a production-quality librarian, such modification privileges might well be restricted to a single user (a library administrator) or at least a small group of users (a library review board).

The sophistication of this drive your-own librarian is limited by the extent of the knowledge engineering that has been done for the domain and the individual components housed within the library. Search and retrieval performance is based on the navigational savvy of the library user as well as the general browsing capabilities over the domain model that are provided by the AdaKNET `Browser_Editor`. This method of interface is more appropriate to particular situations and is not intended for general use.

Classifier Librarian

The classifier librarian utilizes the knowledge base structure itself to lead the user through a dialogue aimed at locating software components that might be of interest. When located at a particular category, there are two sources of additional information that can automatically be inspected — the subcategories which exist at the current category, and the attributes that distinguish one subcategory from the next.

After the user has browsed to a point in the knowledge base and is unable to determine where to go next, or perhaps starting at the very highest level of the knowledge base, choices are offered to the user based on the available branch points that exist from the current node. If the user is able to make a choice based on a name in the list of possible subcategories, the user chooses one and then the classifier librarian once again presents a list of possible subcategories representing where to go next. This process continues until the user is led to the bottom level of the domain network at which point no possible subcategories exist, or else the user can opt out of the automatic descent at which point the browsing mode described previously is automatically re-enabled.

One reason to end this simple mode of descent is that the names of the possible subcategories do not themselves present sufficient information to make a choice. In this case, the user can elect to have the system display the differences between any two categories that were enumerated in a previous list of choices. The user, having examined the differences, can go directly to a category, or else the guided mode can be resumed so that a further descent is conducted in a controlled manner.

Another reason to abort the guided search would be if the list of subcategories to consider did not include any that were likely to contain a software component with the right properties. In this case the user can elect to go to another category to conduct the search from there, or perhaps return several levels back the descent path so that an alternate choice can be made and a new descent path followed.

Finally, the user may locate the proper subcategory before the end of the descent path is reached. This would occur if the user actually found a category that was expected to contain software components with the desired characteristics. Such a category may indeed have subcategories with special properties but these specializations may not be desired. Or, the user may actually intend to create a new subcategory which is to contain some new software components. Once again, domain model modifications (caused by changes to subcategories and their attributes) or even simple software component additions or deletions may be strictly controlled by access privileges so that library integrity can be assured.

Advisor Librarians

The success of a structure-guided tour of the domain knowledge base depends on the richness of the component taxonomy as well as on the fact that a direct path exists from a given point to an eventual stopping point. In a large, complex knowledge base such a path-based search approach is likely to be weak and unwieldy. Additionally, only one segment of the user community is likely to be well-served by a particular taxonomical organization (at best perhaps the "average" user). Different user classes will come to the repository with varying amounts of domain knowledge and navigational skills. For the repository to be useful to a large community of users, its management must support the needs of different segments of the community.

While there can be only one taxonomy in effect at any one time, different levels of support can be provided based upon the class and skill level of the user. Explicit domain "experts" fill the needs of these dissimilar user profiles. Domain expertise, in the form of navigational support and guidance, can be supplied through various advising rule bases implemented with AdaTAU. Like Gaddy, these rule bases are distributed throughout the static domain model represented through AdaKNET. But unlike Gaddy, the services of distributed AdaTAU provide the capability of making long distance jumps within the domain taxonomy.

Advising functionality can play a number of different roles in the effective use of a domain specific repository. Primarily, advising rule bases provide navigational guidance over a complex network and encode heuristics regarding effective retrieval and composition of components in the library. Rule bases can support different project (development, management, QA, ...) as well as domain knowledge (novice through expert). Such diverse classes of users clearly have different aims and needs in using the library, and usage advice should be given accordingly.

Another role is to connect software components to one another that are not directly connected (or are only indirectly connected through several levels within the taxonomy). In the benchmark domain, for example, benchmark programs are useful only when compiled with related timer routines. Taxonomically, these timer components might be quite unrelated to the benchmark programs which are categorized by the feature they test, or the benchmark suite they are located in. Of course, one of the attributes of a subcategory ought to be the category which contains such related pieces of software. But using such conceptual information might be difficult for the novice or casual library user. In such cases, rules can advise the user of the required relations, and even provide the navigational mechanism from one category (or component in a category) to another.

Advising can also help with the task of system configuration from components installed in the repository. Successful subsystems that have been built out of library components can be represented by special purpose rules that connect together these components and provide a navigational path that can be used to recreate the construction of the subsystem. Other system configuration data can be included in the knowledge base to provide factual design and usage information and to point out potential trouble spots.

An important class of rules can be established for trouble-free management of the repository. Library management can be a troublesome task especially for a large and dynamic application domain. Rules can provide guidance for safe insertions of new components into the repository, as well as coverage analysis so that when parts are offered for installation, the library is first examined for related parts and for necessary relationships. Using the benchmark domain as an example, a new feature benchmark should not be installed without an associated timer routine. By allowing library management only under rule-based guidance, the library administrator could in fact be less than a domain expert.

Qualifier Librarian

The quality of parts placed in a library is critical to the success of part reuse. A library framework must provide quality assurance for the parts currently within the library, as well as for parts offered for inclusion within the library. Quality assurance encompasses more than enforcing a testing methodology. Nonetheless, a good place to begin to provide quality measures is to assure a prospective borrower that the part meets some minimal standards and that certain kinds of tests have been conducted on the part.

Gadfly is a knowledge-based testing assistant, built on AdaKNET and AdaTAU, and so an obvious first step is to incorporate Gadfly into the RLF framework. Gadfly's knowledge bases were initially prepared to support simple black box Ada unit testing concentrating on the parameter profiles of a unit under test. For an application-specific domain library, knowledge about the domain itself needs to be added so that such knowledge guides the test analysis and generation process. Gadfly's emphasis on module parameter profiles must also be modified to accommodate its use with general software components which may not have any parameters within their visible external interfaces.

In a library setting, Gadfly can be applied for two different purposes. When a part is offered to the library, Gadfly is invoked to establish the degree and types of testing that need to be done in order that the part be approved for inclusion in the library. In a production-quality version of the qualifying librarian, a part should be officially validated according to the test plan generated by Gadfly before it is officially installed in the library.

Gadfly can also be an option to be invoked by a borrower of a component from the library. In this form, Gadfly seeks to probe the borrower for assumptions regarding the intended performance and error handling characteristics of the module being sought. If the borrower's expectations don't match the module under consideration, an alternate module can be suggested. At the very least, the desired characteristics of the part are compared against the characteristics established during the Gadfly session conducted when the part was installed into the library. The user is not left in the dark about the suitability of the part if this information is captured in the knowledge base itself.

Gadfly just scratches the surface of the problem of quality assurance for software libraries. Just as domain knowledge can be captured to support effective cataloging and retrieval of software components, domain knowledge is the key to providing effective quality assurance for library patrons, as well as for software component authors.

Future Directions

There are many opportunities for growth of the RLF project, and the library framework in particular. Our near term plan focuses on exploring several other application domains and strengthening the Gadfly/Quality Assurance capabilities of the prototype. We can best critique our knowledge collection and representational abilities by trying to build relatively large models over several different application areas with different characteristics. Some examples of such applications areas are real-time, embedded systems (such as missile guidance software), data management systems (such as those required to support military logistics planning and control), and user-interface models such as the X Window System.

Our foremost longer-term desire is to replace the underlying primitive use of the Ada file I/O services with an efficient and powerful data base management system. There is simply too much, and too wide a variety of, data to be processed effectively with raw I/O operations on large numbers of Ada files. Using a DBMS (or preferably an Object Management System (OMS)), we can install both the knowledge base constituents (nodes and attributes in a semantic network, and facts and rules from rule bases) and the software components themselves in a unified database system. Distributed systems are certain to rise in usage and popularity, and by incorporating a distributed DBMS (OMS), we can migrate the RLF naturally to such systems. A system that provides access to a heterogeneous system would be ideal in that knowledge base components could be located on a fast local server, while the components themselves could be stored at various distributed sites served by slower speed servers. A DBMS/OMS would also facilitate the handling of more diverse software artifacts such as requirements, design specs, test cases and reports, etc.

We would like to expand our repository management capabilities. Possibilities include maintaining and reporting usage statistics about component withdrawals, maintaining and reporting reliability measures based on borrower reports about using withdrawn parts in applications, and audit trails that provide library usage information and perhaps lead to the creation of individual user profiles. Such profiles could support individualized start up settings for preferred mode of interaction (browser, classifier, advisor) and could automatically change usage modes as user experience increases.

Finally we would like to expand the native intelligence of the librarian itself. We envision a pro-active librarian that could automatically manage the repository and underlying knowledge bases. Expanded roles and functionality for such a librarian include the rearrangement of components and categories based on usage frequencies and the installation of new components, identification of missing components or library categories which are lacking in quality or numbers of components, and even the actual solicitation of new components based on unsuccessful withdrawal attempts. A related long-term goal is to incorporate the use of generation techniques so that a tightly coupled collection of software components could themselves be replaced by a generator and a much smaller collection of specifications. Each component in the family could be produced by the generator from the proper

5. Experiences With Ada

Various Ada features have been positive factors in the design, coding, implementation, and integration of the knowledge representation systems and applications developed during the RLF project. Certain aspects of the current generation of compilers and run-time systems posed obstacles that needed to be overcome. This section presents some of our basic "lessons-learned".

Our approach to the construction of AdaKNET and AdaTAU emphasized the basic design principle of data abstraction and depended heavily on the use of Ada generics and exception handling. By concentrating on a careful and reasoned use of Ada capabilities, we were able to develop systems with traditional AI functionality that also promote system maintainability and evolvability. Our plan for growth and evolution of our systems includes analysis of various feature sets that are deemed useful within possible configurations of deployed systems. This *domain-analysis* sets the stage for early system versions that, while not fully equipped, are potentially useful in applications that do not require the entire complement of features. Thus prior domain analysis leads to an orderly evolution from limited prototypes to a final product. In fact, such early systems are also more compact and more efficient than their full-featured counterparts and so may be appropriate where there are resource constraints of one type or another.

Careful design also affords significant opportunities for reuse at the module/package level as well as the sub-system level. For example, during the design of AdaKNET, we identified the need for a tables abstraction that simultaneously provided persistence for the objects and relationships being managed by AdaKNET as well as for efficient access to stored data. Later, when the implementation of AdaTAU required a means for AdaTAU structures to be made persistent, the tables generic provided the necessary functionality. Our experiences with Gadtly and the various librarian applications are ample proof of the reusability of the AdaKNET and AdaTAU subsystems themselves.

Reuse of another, perhaps more powerful sort, is provided by the use of generation techniques. From our high-level knowledge base specifications, we automatically generate Ada code to initialize machine-processable knowledge bases. Because the specification language processors are themselves generated from their own high-level specifications, we need not hand-write code to create parsers and lexical analyzers for these languages. The small amount of handwritten code that is needed to support the semantic processing provided in the language is often an easy adaptation of code provided for other languages that were developed using the Ada-based compiler-compiler technology developed here at the Paoli Research Center.

The adaptation of the RLF knowledge base components to new application domains is achieved in part by writing high-level specifications of the knowledge that captures the new domain and then automatically generating the initialization code. Only a relatively small amount of application-specific code needs to be handwritten. Another advantage of the use of specification languages is that knowledge descriptions can be written by domain experts who are not necessarily Ada experts.

Our collective project experiences were not totally positive, however. Many of our difficulties can be traced to problems with the Ada compiler and run-time systems that we employed during the project. We faced problems with reuse of outside software components, execution performance of our own code, and a less-than trouble-free port from a workstation platform to a PC platform. For the most part, the blame for these problems cannot be attached to Ada, but rather to our own inexperience and to the limitations present in today's Ada language systems.

Our foremost problem was a profound lack of control from within our system of dynamic memory — memory that is allocated through the use of the Ada new command. Although we attempted to be as careful as we could possibly be regarding memory management, returning memory back to the memory pool (heap) when no longer needed, we could not control process memory growth using completely portable Ada code. A large part

of the problem turned out to be the Ada run-time's own use of the heap for I/O buffers among other things. Because the application's use of the heap was conflicting with the system's own use, our applications invariably experience large-scale heap fragmentation and subsequent performance degradation. After talking to several Ada vendors, it became clear that the only way to completely control heap memory was to take matters into our own hands and write code specific to the compilation system in use, thereby *a priori* sacrificing portability.

An interim solution turned out to be as simple as providing a limited amount of memory management from within each basic abstraction. For example, object free lists are managed from within each of the basic abstractions such as lists, sets and tables. Whenever such objects are allocated (using *new*), they are never released using unchecked deallocation, but rather are attached to the free list when the object is logically no longer needed. Our experience to date is that though process memory does creep upwards as execution continues, this growth is tolerable, and performance is not severely affected. One very important lesson-learned is that for applications requiring extensive use of dynamic memory, heap management is not a luxury, or a performance related fine-tuning, but rather a necessity.

In addition to less-than-perfect memory management (e.g. file buffers are allocated on the heap, but are never deallocated after the file is closed), we also experienced a number of other compiler system problems. Both compiler systems we used exhibited Ada library management and performance problems as well. We had problems involving the use of generics (generic instantiation glitches and compile-time performance problems with nested generics), exceptions (poor propagation performance), and the use of Ada-Make facilities which are supposed to automate the process of generating executables after changes to various source files. Today's generation of Ada compilers require large amounts of disk space to support their custom forms of library management. Executable object files also tend to be very large (certainly compared to other less powerful languages). We expect many of these problems to disappear as compiler vendors improve their products.

Finally, although portability is aided by Ada, it is not assured, nor is the process made trivial. When porting our system from one compiler to another on the same hardware, we experienced difficulties due to library management differences as well as compiler limitations. When porting from one machine to another, using the same compiler we once again ran into compiler limitations, as well as resource limitations (in a PC environment). We also experienced minor, but tedious, operating system consequences. For example, due to our use of text files to store knowledge bases, the port to an environment with very strict file naming conventions was not as easy as expected.

6. Summary

Some idea of the scope and size of the project is obtained by examining the development platforms that were used in working on the project and some statistics that summarize the large amount of code and documentation that have been produced. Our primary development environment consisted of Sun Microsystems workstations (3/50, 3/60 and 3/75's) each of which contained a minimum of 4 megabytes of memory and was operated under Sun operating system 3.4. Available Ada compilation systems included Verdex Ada Development Systems (VADS) 5.5 and Alsys 3.2. Our initial development took place using VADS mainly because of our initial familiarity with this system. As work proceeded, we endeavored to use the Alsys compiler to remove any system specific dependencies. To demonstrate machine portability, as RLF components were completed and refined, the source code was transferred to a Unisys PW7/500 computer (IBM compatible, using MS-DOS 3.3) and compiled under the Alsys compilation system which includes an extended memory board.

The system, at the time this paper is being written, is comprised of 70 handwritten Ada packages (23,000 Ada statements, 80,000 lines including comments) and 84 generated packages (for the language translators) containing approximately 13,000 Ada statements. Knowledge base descriptions (RBDL and SSDL specification files) contain over 6,000 clauses (each clause roughly equivalent to an Ada statement). Over 500 pages of documentation have been produced during this project. This documentation includes design reports, user manuals (both as on-line ASCII text, and formatted printed copies) and presentation materials from several conferences including TRI-Ada '88 and AIDA '88.

The basic achievement of the RLF project is the provision, in a demonstrable prototype, of a general framework for the construction of domain-specific libraries of Ada software components. The domain-models expressible using the RLF components support the statement and evolution of detailed taxonomic descriptions and supportive search capabilities that enable high-impact reuse. In addition, we provide automated support for users of reuse libraries including component selection, insertion and qualification.

An important incremental achievement was the successful application of knowledge-based technology to the area of Ada component testing (Gadfly). Both of these achievements were accomplished based on the production of general-purpose knowledge-based components in Ada. These tools themselves support the incremental addition of knowledge-based processing to other Ada products and systems. The project also represents a successful experiment in the hybridization of dual knowledge-representation systems that together insure robust coverage of procedural and declarative forms of information.

Finally, we were able to experiment with innovative uses of Ada for the implementation of knowledge-based tools. We have learned a great deal about the techniques and approaches that support the design of reusable software components and believe that this knowledge can drive further work on the RLF project as well as guide us as we develop our own component reuse libraries.

References

- BAR85** "Domain-Specific Automatic Programming", *IEEE Transactions on Software Engineering*, November 1985, pp. 1321-1336.
- BAT83** J. C. Batz, P. M. Cohen, S. T. Redwine, J. R. Rice, "The Application-Specific Task Area", *IEEE Computer*, 16 :11, pp. 78-85, November 1983.
- BRAC81** R. J. Brachman and J. G. Schmolze, "An Overview of the KL-ONE knowledge representation system", *Cognitive Science*, 9(2):171-216, 1985.
- BURT87** B. A. Burton, R. W. Aragon, S. A. Bailey, K. D. Koehler, L. A. Mayes, "The Reusable Software Library", *IEEE Software*, July 1987, pp. 25-33.
- FREE83** M. W. Freeman, L. Hirshman, D. P. McKay, M. S. Palmer, *KNET: A Logic-Based Associative Network System*, presented at the third Knowledge Representation Workshop, Santa Barbara, CA, October 1983.
- MATU88** P. Matuszek, J. Sable, J. Clark, D. Corpron, D. Searls, *KSTAMP: A Knowledge-Based System for the Maintenance of Postal Equipment*, Third Annual United States Postal Service Advanced Technology Conference, May 1988.
- MCCA86** Ron McCain, *Reusable Software Component Engineering*, IBM FSD, Houston, Texas, July 1986.
- MCIL76** M. D. McIlroy, "Mass-produced software components", in *Software Engineering Concepts and Techniques*, 1968 NATO Conf. Software Eng., J. M. Buxton, P. Naur, B. Randell, Eds. 1976, pp. 88-98.
- PAYT82** T. Payton, S. Keller, J. Perkins, S. Rowan, S. Mardinly, "SSAGS: A Syntax and Semantics Analysis and Generation System", *Proceedings COMPSAC '82*, November 1982.
- POLL87** R. Pollack, W. P. Loftus, J. Solderitsch, *A Generative Approach to Message Format Processing*, presented at CASE '87, May 1987, Boston, MA.
- PRIE87** R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability", *IEEE Software*, January 1987, pp. 6-16.

SIM088 Mark Simos, *The Growing of an Organon: A Hybrid Knowledge Based Technology and Methodology For Software Reuse*, presented at the 1988 NISOP Conference on Software Reusability.

WALL88 K. C. Wallnau, J. J. Solderitsch, M. A. Simos, R. C. McDowell, K. A. Cassell, D. J. Campbell, *Construction of Knowledge-Based Components in Ada*, in proceedings of AIDA-88, George Mason University, November 1988



James Solderitsch is the chief programmer of the STARS Foundations reusability Library Framework (RLF) project. Dr. Solderitsch joined the Unisys Defense Systems Software Technology Laboratory in January 1986 after having been an assistant professor of Computer Science at Villanova University for 8 years. His primary interests are software reusability and the impact of very high level, domain-oriented, specification languages on software productivity and reusability. He holds a B.S. degree in Mathematics from Villanova University and an M.S. and Ph.D. degrees in Mathematics from Lehigh University.



Kurt C. Wallnau served as team leader on the RLF project responsible for the design and implementation of AdaKNET, the semantic network knowledge representation component of the RLF. Kurt Wallnau has several years experience in the design and prototyping of Ada environment tools. His key research interests are in object-oriented systems, integrated software environments and advanced data modeling techniques in software engineering environments. He holds a B.S. degree in Computer Science from Villanova University.



John A. Thalhamer is the manager of STARS Foundations projects at Unisys. He has over nine years experience in the design and implementation of software support tools. His primary interests are in the areas of test/validation tools, Ada, compilers, and user interfaces. He holds an M.S. in Computer Science from Cornell University and a B.S. in Computer Science from the Pennsylvania State University. He is a member of the ACM and the IEEE Computer Society.

Ada, Hypertext, and Reuse

Larry Latour

Computer Science Department
University of Maine, Orono, Maine

Abstract

In this paper we discuss hypertext as a tool for describing taxonomies of Ada packages in order to facilitate their reuse. Our basic premise is that the primary inhibitor to the reuse of software components is understanding. We describe a component as an information "web" of attributes, containing specification, implementation, and usage information. The hypertext model is used to describe component information webs, alternate taxonomic structures leading to these webs, and class information webs describing information common across component webs.

Introduction

Reusability is a noble goal to shoot for in a software engineering environment. Its advantages are obvious. Development costs can be decreased by reusing code, algorithms, module specifications, subsystem designs, etc. Reliability can also be improved by reusing well-engineered, well-documented, well-tested parts from a parts library. In fact, it has been argued that the advantage of increased reliability alone justifies reuse, even if development costs might be increased.

Unfortunately a number of inhibitors exist that tend to negate the advantages described above. The amount of work required to find a pre-written part (whether in a repository or book), to decide whether or not it is applicable to the problem domain, and to integrate it into a system might more than offset the savings gained from not having to develop the part. In addition, the "shoe-horning" of a pre-written part into a system might lead to reliability problems at the system integration level.

A good deal of work has been done on the organization and retrieval of reusable software components [2]. Unfortunately,

little work has been done on the evaluation problem, i.e., the understanding problem. Assuming that components stored in a library are well-designed for reuse, and assuming also that there exists a way to organize these components for retrieval, the problem still remains as to how the components will be evaluated once retrieved. There is usually a wealth of knowledge that must be considered when performing such an evaluation, and little consideration has been given to tools to aid in this "understanding" process. One such class of tools, based upon the hypertext model, seems to be the right approach to the problem.

Hypertext tools allow the user to create webs of annotated nodes and links, manipulate information (textual, graphical, voice, etc.) within the nodes and links, and navigate the webs by various textual and spatial commands. At UMaine we have constructed the hypertext-like system SeeGraph [6], in Ada, on our Vax workstations, and have also made heavy use of Hypercard [1], a Macintosh tool providing a good deal of hypertext functionality. We have used SeeGraph to store and retrieve the knowledge web of information embodying the SeeGraph system itself, and are using Hypercard to construct the knowledge webs surrounding both the taxonomies and component parts of the Ada Booch Components [3] and McDonnell-Douglas CAMP parts [4].

Hypertext is not in itself a solution. One must be careful, when constructing hypertext webs, that their structure does not get out of hand. When using ill-defined hypertext structures I am reminded of the hacker's game of Adventure. I recall entering the room of a friend's ten year old son, being amazed to see the complete map of the Adventure "world" attached to a good portion of his wall. Without such a map, it is almost impossible not to get lost in this "world". The same thing tends to happen in hypertext.

Hypertext is to information structuring as goto's are to programming. Early writers of programs using sequence and transfer of control operations must have said, "What wonderfully complex systems we can construct - systems to control robots, guide missiles, balance our checkbook! ... Amazing!". We must view hypertext in this same manner. Methods must be developed that parallel the systems design notions of structured programming, information hiding, and

abstraction. This, to anyone forced to build such webs, should be obvious.

We have discussed the SeeGraph system in [7]. In this paper we concentrate on our experiments in Hypercard. We show the structure and content of component information webs as well as the flexibility of the hypertext model in allowing for the construction of alternate taxonomic structures of components and sharing of component attributes. We finish by discussing the usefulness of Hypercard as an application generator, and the implications of this for reuse.

A Note about the Example Component Libraries

In our study we have made use of both Grady Booch's component taxonomy and the McDonnell-Douglas CAMP (Common Ada Missile Packages) components. Booch's component collection includes approximately 500 Ada "computer science domain" components, including a number of variations of stacks, queues, graphs, sorts, etc. The McDonnell-Douglas collection consists of approximately 200 Ada components, containing both computer science domain packages similar to Booch's as well as components particularly suited to missile guidance system software.

Components as Information Webs

The problem with many component libraries is that the code is considered to be the component when for purposes of understanding it is only one attribute of the component. Generally there exists (or should exist) a significant body of knowledge about a component, broadly divided into three categories - specification information, implementation information, and usage, or domain, information. This information exists as a complex web of facts about a component that must be well understood in order to use it properly. In fact, it can be argued that this information web is the component, one small part of which is the code. Furthermore, to the extent that there is a one to many mapping of

specification to implementation, a component could be considered a specification together with a class of implementations, each requiring its own unique mix of resources (e.g., memory, external storage, processor time, task overhead, etc.).

Typically, component information takes the form of printed documentation. Unfortunately, printed media suffer from the drawback of sequential organization. A good example of this is the Ada Reference Manual. Embedded within this manual is a web of reference trails that more often than not creates an uncomfortable maze for the Ada programmer. Hypertext tools provide a much more natural mode of access to these information webs.

When considering specification, implementation, and usage information more closely, one quickly becomes aware of the breadth of information that can be associated with each module. We do so in the following sections.

Specification Information:

The specification is the contract between the user and the implementator, and as such needs to be as precise, yet as simply stated as possible. This information might include an Ada syntactic specification, a descriptive and/or graphical presentation of the user's mental model of the specification semantics, formal semantic descriptions, and syntactic and semantic justification.

For example, consider the web of specification information for Booch's undirected unbounded unmanaged graph, represented by the Hypercard "card" in figure 1. Using this web, the user can access specification information in a variety of forms.

Hypercard allows the user to create cards and place any number of "buttons" on a card. These buttons provide an area on the card for the user to "click on" with a mouse, invoking a procedure which can transfer control to another card as well as perform any number of housekeeping chores. Clicking on the mental model button, a combined text/graphical description of the package appears (figure 2). The purpose of this information is to give the user a quick and dirty feel for the overall behavior of the package. It is a rough sketch of package semantics.

The Ada spec. button leads the user to a syntactic description of the package interface (figure 3). This is a minimal, compiler readable form of the interface,

providing semantics only in the form of operation names, formal parameter names and types, and exception names. Notice that Hypercard allows for the creation of scrollable fields, as shown by the scroll bar to the right of specification.

The informal spec. button leads to an Ada specification annotated with english language semantic descriptions in a form similar to that used by Guttag and Liskov in [5] (figure 4). From the effects clause of the ADD operation, we see that a vertex is created, an input item is placed "in" the vertex, and the vertex is added to the graph. The restraints clause defines any requirements on the input parameters, the modifies clause defines which if any of the input parameters are modified on procedure completion, and the exceptions clause defines the Ada exceptions that might be raised by the procedure along with the abnormal effects of the procedure if such exceptions are raised.

The Formal spec. button leads to a formal larch-like [5] interface specification (figure 5). The semantics of this specification are defined in terms of an algebraic model of graph semantics, accessed through the Formal model button (figure 6). This two-tiered approach to describing abstract data type behavior is useful in that it allows for the separation of language specific features such as procedure side effects and exceptions from language independent semantic descriptions written in a straightforward functional style.

In addition to the buttons leading to overall semantic descriptions, there is a button leading to a justification of specification syntax, and a button leading to a description of the operations in the interface that collectively form an iterator. The justification description includes a rationale for why this particular syntax was chosen along with examples of other possible forms that could have been chosen. In our particular prototype the iterator button leads to a single card describing the iterator operations and presenting a graphical view of the iterator. It is important to note however that the flexibility of Hypercard would allow us to represent the iterator by yet another "web" card containing buttons leading to a different attribute of the iterator.

The remaining buttons are traversal buttons, i.e., they aid the user in moving to related "web" cards. The Graphs button leads back to the Graph class state window (see taxonomies sections), from which the user can choose another graph and begin the exploration process anew. The Usage

and Implementation arrow buttons lead the user to web cards for the respective information.

Implementation Information

The implementation is the satisfaction of the specification contract, and the user needs to be comfortable that it indeed accomplishes its goal. In addition, the state of the art in specification is such that the implementation must be there to "fill in the blanks". This information might include a properly structured and annotated Ada body, module design information, a mental model of implementation structures, verification information, and implementation tradeoffs.

Consider the web of implementation information for the graph specification described in the previous section (figure 7). This implementation web includes both english language and graphical descriptions of the representation of a graph, an Ada body, the Ada private part of the specification, verification and test plan information, the resources used by the implementation, and a justification of the package implementation design. Notice that the Ada private part, although physically in the package specification, is actually part of the implementation, and should be treated as such. This is not really a problem, as environment tools can present two different views of an Ada package specification, one containing the private part for the compiler, and the other minus the private part for the user.

Usage Information

Usage information is an extension of the specification semantics. In the case of complex interfaces, the user needs to understand how the functionality of a component can be combined to construct a complex application.

As mentioned earlier, a specification is a contract between the user and implementor of a component. Complicating matters here is the promotion of reuse as a software engineering goal. Clearly a specification needs to be designed with a class of users in mind rather than just a single user. The usage information supplied for a particular component therefore should contain not only specific examples of component usage, but also the characteristics of the class, or domain, of users that can use the component, and general patterns of usage within this domain.

The above information is to a great extent domain specific, and it is not entirely clear to us how much information to supply for a general purpose "computer science domain" package such as Graph. For our prototype system we have provided a number of examples, including one for a PERT chart (figure 8), as well as general patterns of Graph usage as typically presented in good data structures texts.

Representing Taxonomic Structures:

In addition to describing component information webs, hypertext has proven useful for describing taxonomic structures of components, as well as information associated with component classes. For example, the class state window in figure 9 is used to access the class of graph components in the Booch library.

The current "state" of this card is of an undirected unbounded unmanaged graph. In this state the info web button leads to the associated component specification information web.

To change the card state, the user needs simply to click on (thus highlighting) the Change state button, followed by any combination of the Directed, Bounded, and Managed buttons.

The Modules button leads to a card describing a broad class of computer science modules in the Booch components, including the Graph class in particular. The ? button allows the user to exit the Hypercard application to start anew.

"Meta-information" about the taxonomy itself is important for beginning users of the taxonomy. When the Change state button is in non-highlighted mode, the user can click on any node in the tree and get information about the meaning of that node. Specifically, the Graph button leads to a class information web, discussed in the following section.

Exploring Commonality

As we have constructed knowledge webs around the individual component parts of both the Booch Components and CAMP parts, we have become aware of a great deal of information shared across modules, the sharing corresponding to the classification criteria used in the taxonomies. This has led us to look at the relationships across information webs within various component taxonomies.

Consider the following examples. Test plans and test drivers are similar across classes of components - stacks,

queues, graphs, etc., and implementation models are similar across components with the same form. Algebraic models are also similar across classes of components - the sequence being a possible model for stacks, queues, deques, etc.

Recall that the Graph class state window contains a Graph button. The intent here is that this button lead to a class information web. The structure of the class information web reflects the similarities between the individual information webs of that class. For example, the class information web contains a mental model button, describing the general attribute-independent model of a graph, and a formal model button, describing the formal model of a graph used by all components of that class.

Our aim here is for the user of a taxonomy to be able to extract as much information as is possible without having to commit to a particular component, thus further speeding up the evaluation process.

Shared Resource Webs

In addition to experimenting with class information webs, we have experimented with tying component information webs to information webs about a particular web attribute. One area we are particularly interested in is formal specification.

We have taken the Larch handbook of algebraic models and have created a hierarchically structured hypertext web of specifications, including sequences, graphs, stacks, etc. We have then attached component information webs to this specification "database" via the formal model button. The user can then access the formal model of a graph from this shared resource. In addition, the backtracking capability of Hypercard provides a trail for the user to back out through once the formal model web is entered.

Alternate Taxonomic Structures

At times it may be valuable to provide alternate access to a collection of modules by means of different taxonomic structures. For example, rather than using the Booch taxonomy to access his components, we may prefer to use the faceted classification method of Ruben Prieto-Diaz [8]. With hypertext, we can provide both structures in a single hypertext web.

As mentioned earlier, we have also applied the hypertext model to the CAMP (Common Ada Missile Packages) components of McDonnell-Douglas. While the general CAMP taxonomy is a fairly rigid hierarchy, as pictured in the Hypercard card of figure 10, alternate means of package access exist. Specifically, an expert system has been developed by McDonnell-Douglas where, through a series of questions and answers, the user is led to proper choice of a component in the collection. Hypertext can be used as the information structuring language in connection with such a system. The user then has a choice of whether to be guided to a choice or to peruse the taxonomy independently.

Hypertext Systems as Applications Generators

A number of hypertext systems, Hypercard included, contain a rich set of tools for prototyping user interfaces. It was for this reason that we began experimenting with Hypercard. Our intention was originally that Hypercard be a "stop-gap" until we completed our SeeGraph system, but hooks in Hypercard along with the availability of Macintosh-DEC interfaces make a full-scale library system with a Hypercard front-end feasible.

Hypercard, with its flexible graphics editor, script language, and built-in facility to create "stacks" of cards, contributes significantly to reuse. A number of Application Generation tools such as this one exist under the headings of CASE tools, 4GLs, UNIX tools, etc., and it is in these areas that we are making great strides in reuse. As with the accessibility of component libraries described earlier, understanding is the key issue. Whether we provide parts for systems designers to compose systems or application generators to generate systems, the designers must be provided with a clear understanding of what is provided. This is a noble goal that is rarely achieved.

Summary

Our premise in this paper is that understanding is the primary inhibitor to the reuse of software components. We have used the hypertext model to describe information webs of components, collections of interrelated component attributes that we argue collectively define a component.

A number of interesting uses of hypertext emerged from the construction of the above webs. We recognized that we

could construct and describe taxonomic structures of component collections using the same hypercard model as we used to describe the components themselves. In addition, we could easily construct alternate taxonomic structures for the same component collection, class information webs representing classes of component information webs, and shared resource webs describing "databases" of information for each component information web attribute.

A large part of constructing the information webs for component libraries such as those described in this paper is tedious work. We have made an effort to fill out the webs for both the Booch and CAMP modules in a way that "proves the concept", but much work needs to be done to complete the project. A good deal of information already stored is repetitive and needs to be consistency checked, and a good deal of study needs to be done to eliminate existing redundancy. We are currently looking toward techniques developed in the object-oriented programming world to restructure our library components in this regard.

Acknowledgments

The author gratefully acknowledges the work of Elizabeth Johnson and Carol Roberts, graduate students in the Computer Science Department. Ms. Johnson spent long hours constructing SeeGraph on the Computer Science Dept's Vaxstation cluster, and Ms. Roberts spent long hours constructing the Hypercard prototype, populating the Booch module webs with information reconstructed from his packages and textbook.

This work is partially funded by the CECOM Center for Software Engineering under contract No. DAAL03-86-D-0001, Delivery Order No. 1041, Scientific Services Program.

References

- [1] Apple Computer, Inc., Hypercard User's Guide. Cupertino, CA: Apple Computer Inc., 1987.
- [2] Biggerstaff, T.J., and Richter, C., "Reusability Framework, Assessment, and Directions", IEEE Software, March 1987, pp. 41-49.
- [3] Booch, Grady, Software Components with Ada, Benjamin Cummings Publishing Company, Inc, Menlo Park, California, 1987.

[4] CAMP, Common Ada Missile Packages, 3 Volumes, McDonnell Douglas Astronautics Company, St. Louis, MO.

[5] Guttag, John V., Horning, James J., and Wing, Jeannette M., "The Larch Family of Specification Languages", IEEE Software, September, 1985.

[6] Latour, Larry, and Johnson, Elizabeth, "SeeGraph: An APSE Tool for Organizing Reusable Abstractions", Internal Report, University of Maine Computer Science Dept., Orono, July 1987.

[7] Latour, Larry, and Johnson, Elizabeth, "SEER: A Graphical Retrieval System for Reusable Ada Software Modules", Proceedings of the IEEE Conference on Ada Applications and Environments, Manchester, NH, May, 1988.

[8] Prieto-Diaz, R., and Freeman, P., "Classifying Software for Reusability", IEEE Software, January, 1987, pp. 6-16.

About the Author

Larry Latour received the B.B.A. Degree in Statistics from Baruch College/CUNY in 1973, the M.S. degree in Operations Research from Polytechnic Institute of New York in 1978, and the Ph.D. degree in Computer Science from Stevens Institute of Technology in 1985. He is an Assistant Professor of Computer Science at the University of Maine, Orono, ME., 04469. His research interests include database transaction systems, software engineering, formal specification, programming language environments, and reusability.

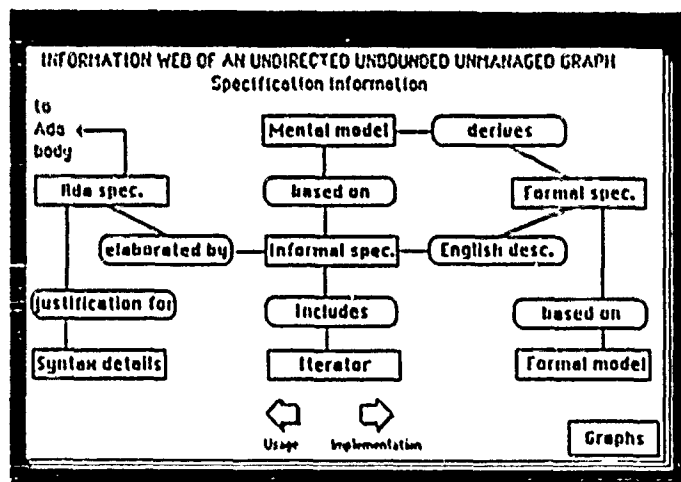


Figure 1

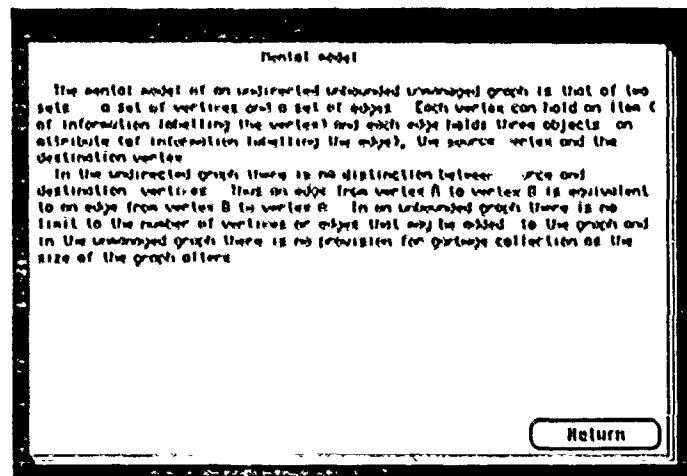


Figure 2

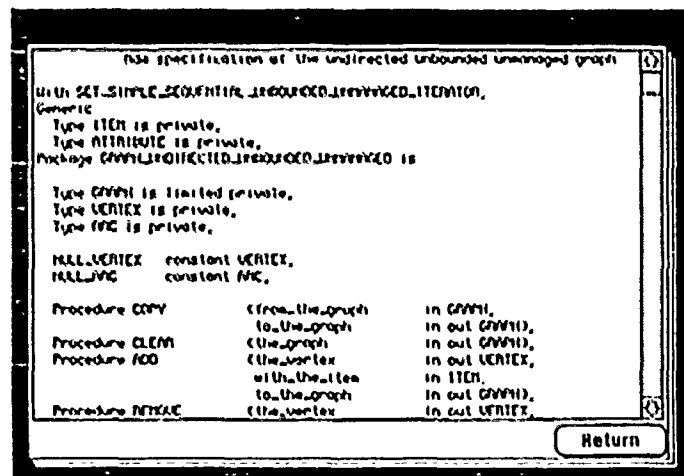


Figure 3

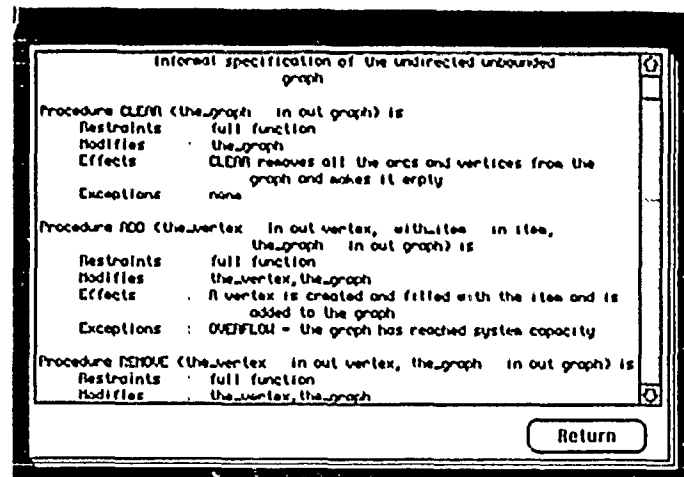


Figure 4

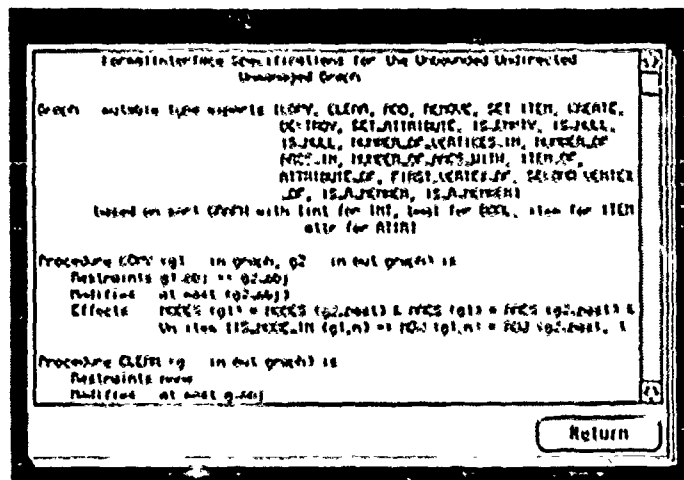


Figure 5

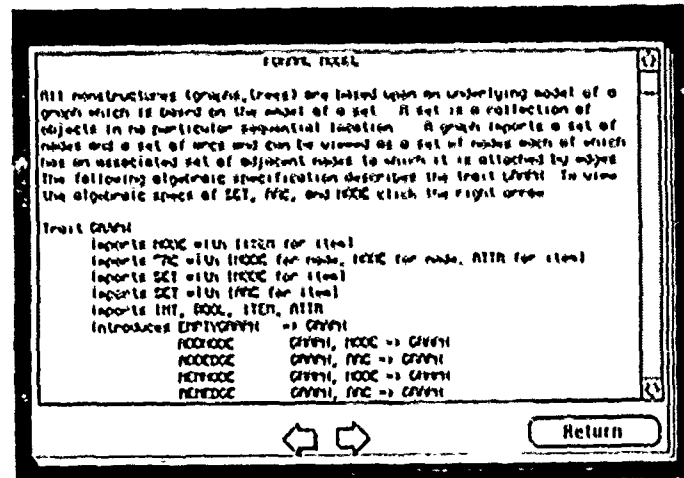


Figure 6

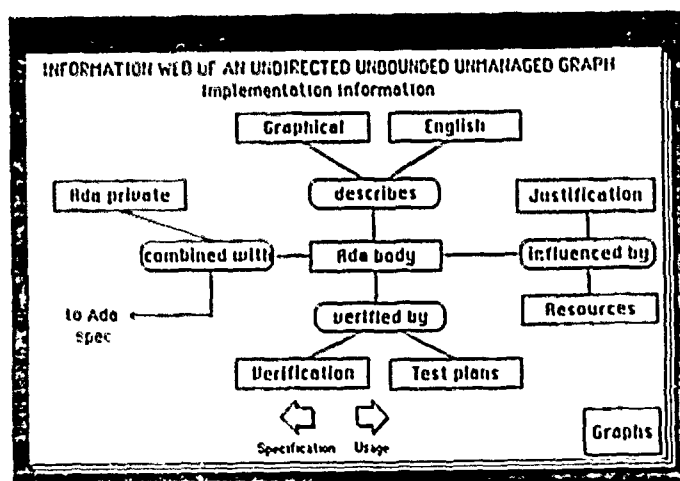


Figure 7

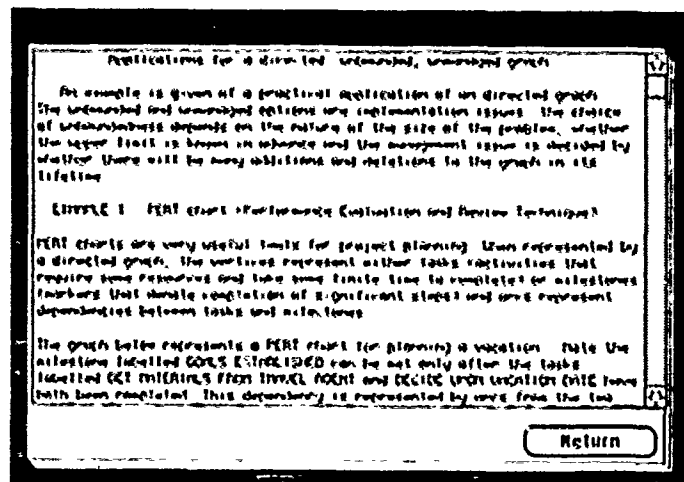


Figure 8

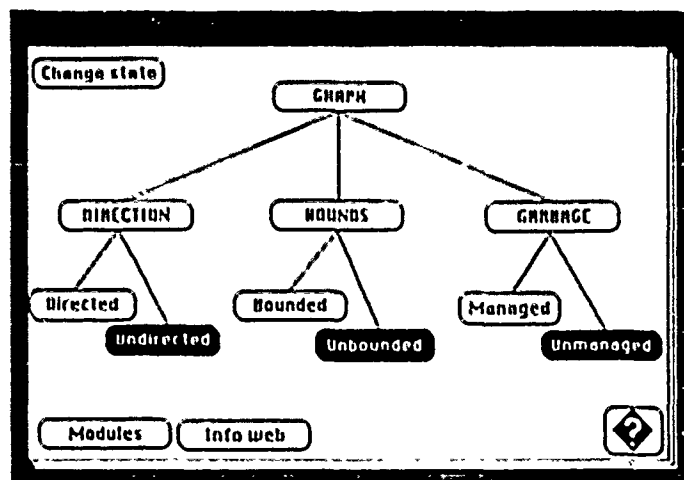


Figure 9

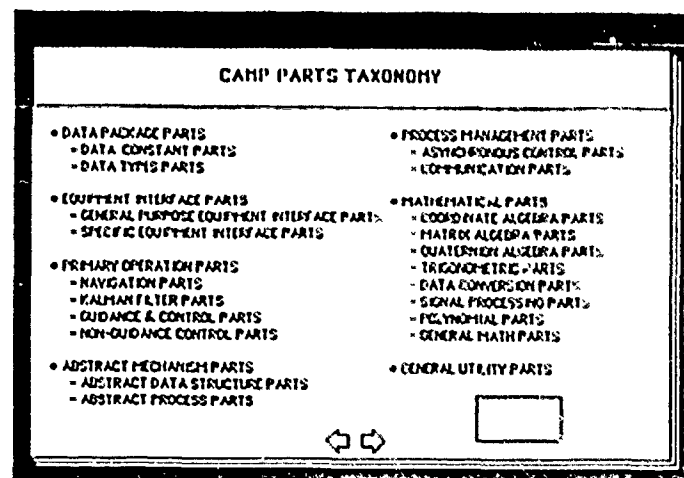


Figure 10

DISCIPLINED REUSABLE ADA PROGRAMMING FOR REAL-TIME APPLICATIONS

Frank Arico
and
Anthony Gargaro

Computer Sciences Corporation
Defense Systems Division
Moorestown, New Jersey

Summary

Many DoD systems have been developed using reusable software parts. Conversely, embedded real-time systems, have been developed with a minimum of software reuse. The inherent difficulties in developing reusable parts for embedded real-time systems result from critical timing constraints or dependencies on processing resources. To overcome these difficulties, an aggressive policy for part reuse must be practiced throughout the software development lifecycle. This paper identifies issues in writing reusable parts within the conceptual framework of real-time programming. A programming discipline is proposed that would be based upon guidelines, paradigms, and uniformity of Ada runtime systems.

Key Words. Ada, real-time programming, software reuse.

Background

The number of Department of Defense (DoD) applications using the Ada language is increasing rapidly fostered by the availability of over 200 base and derived validated compilers covering a broad spectrum of computers. Reports documenting these applications indicate that many have been successfully developed either using reusable parts or having yielded reusable parts. In addition, specific applications have been targeted to demonstrate the efficacy of reusing Ada parts for embedded real-time DoD applications, namely, the Common Ada Missile Package¹. To date, embedded real-time DoD applications have not achieved a high degree of part reuse because of the difficulty of developing reusable Ada code that must satisfy hard real-time constraints intrinsic to deadline-driven systems. This difficulty

is compounded by the emerging use of multiple computers to achieve parallel program execution within embedded real-time applications.

A significant amount of material has been contributed to the literature on developing reusable Ada parts². This material has included guidelines and paradigms for both design and coding; however, there has been limited attention to part performance efficiency and real-time constraints. The exception has been recognizing that performance enhancing dependencies upon the individual execution-time behavior of Ada runtime systems must be avoided when developing parts for mission critical computer resource applications³.

This paper focuses upon issues that must be addressed by a discipline for reusable Ada programming in real-time applications. This discipline combines current Ada reusability programming practices with techniques that remove some of the program execution unpredictability that can thwart writing programs for real-time applications. In addition, the discipline promotes deducing a program's validity from its static text. Programs are designed based upon a precise specification of the execution time constraints of the application together with an understanding of the impact of the Ada runtime system on software reuse⁴.

Initially, the paper presents a brief discussion of the conflicts and difficulties of writing reusable real-time Ada software. The discussion provides a context for bounding the issues that arise from translating into Ada a simple paradigm frequently used by real-time applications. Finally, the paper outlines a proposal for introducing a discipline to practicing programmers.

Conflicts and Difficulties

Explicit in the design goals for the Ada language was a desire to reduce the cost of writing embedded real-time application software. The language's ability to compose an application from independently produced software implicitly advocates software reuse as a significant contribution towards achieving this goal. Unfortunately, the availability of a programming language, however rich in abstractions for software reuse, is insufficient to guarantee reusable code. The lack of guaranteed reusability is particularly evident once adherence to execution time independence is compromised by assumed explicit or implicit processing resources. In the presence of hard real-time constraints, execution time dependencies must be reconciled with software reuse, thereby presenting numerous conflicts and difficulties. These conflicts and difficulties can be ameliorated only through disciplined programming that carefully legislates against the uncontrolled use of implementation dependencies that bound the semantic fringes of the language.

Real-Time Software.

Real-time software generally refers to software parts where correctness depends upon timing constraints over which there may be little or no programmatic control. For example, a part must meet a hard deadline or else it is in a failure mode. Often these dependencies are associated with controlling or monitoring external devices whose successful operational performance is time critical. These dependencies lead to several distinguishing features that may be used to characterize real-time software. Among these features are concurrent execution, event synchronization, fault tolerant execution, and low-level hardware interaction. In this paper, concurrent execution and event synchronization are considered the features that frequently dominate the design and coding of real-time programs, which in turn, compromise the reusability of its constituent parts.

Concurrent Execution. A principal reason for concurrent execution is to decrease the execution time of a program. Parts of a program may execute on different processors or may share a single processor. Consequently, the partitioning of an application for concurrent execution requires the careful analysis of part

dependencies to ensure the optimal utilization of processing resources. Traditionally, the predominance of single processor computers has permitted only the logically concurrent execution of programs. The scheduling of processing resources, or processor sharing, among program parts has become an important dimension of design and coding real-time programs. While multiprocessor computers and multicomputers are now commonplace, many applications do not have sufficient resources to avoid some degree of logically concurrent execution. In addition, there are often instances where logically concurrent execution may be advantageous when physically concurrent execution incurs substantial penalties in sharing data among the parts resident on remote computers. Therefore, scheduling processing resources in real-time programming will continue to complicate part reuse.

Event Synchronization. Event synchronization allows program execution to be formulated with respect to a consistent specification of a part's timing dependencies through a uniform abstraction of time. When the timing dependencies are not predictable, i.e., events are asynchronous, concurrent execution of a part is commonly employed to achieve the necessary event synchronization for correct program execution. Real-time applications often comprise parts whose execution is referred to as periodic, aperiodic, or sporadic. These applications have requirements for synchronizing with differing events, e.g., an internal clock tick, a signal from an external device, or exchanging messages among concurrently executing parts.

Ada Real-Time Model.

In Ada a unified approach to concurrent execution and event synchronization is specified through the Ada tasking model. While the model provides potentially reusable abstractions for concurrent execution and event synchronization, it has been the subject of intensive debate when applied to real-time applications⁶. Three key issues become apparent when writing reusable Ada real-time parts. These are briefly addressed in the following paragraphs with regards to justifying a programming discipline.

Abstraction. The Ada tasking model supports abstractions for the asymmetric

communication and synchronization among autonomous parallel threads of control within a single program. The abstractions can be used to express many classical paradigms that protect shared data and message passing with minimal regard to the underlying processing resources available. While this level of abstraction may be perceived as promoting software reuse, it can be a disadvantage when writing parts subject to critical timing constraints.

Disadvantages result from the unintentional misuse of abstractions and increased semantic complexity introduced into a part whenever concurrency and synchronization constructs are integrated within a programming language. For example, timing anomalies can typically, but surprisingly, occur when insufficient attention is given to task activation and task termination. This type of problem may be obviated by guidelines that recognize the program-wide implications of the abstractions. A more difficult problem is manifested when arcane use of the abstractions becomes necessary to program commonly accepted real-time processing models. For example, the partitioning of a program for distributed execution remains unspecified within the tasking model and leads to a diversity of execution models that are not conducive to writing reusable parts. Therefore, it becomes necessary to adopt rules that facilitate the composition of distributed programs from artificially constructed abstractions or paradigms.

Eventual resolution of these kind of problems may require specification of carefully defined auxiliary abstractions that complement the existing Ada tasking model⁶. The auxiliary abstractions would be used to construct reusable packages that implement common real-time processing models.

Dependencies. The abstraction of concurrent execution and event synchronization into the Ada tasking model would seem to remove many of the transportability obstacles to reusing parts from applications that have traditionally depended upon specialized real-time executives. However, when critical timing constraints are present, parts may become dependent upon specific implementation options permitted by the model or upon constructs having implied temporal semantics. For example, a delay of zero may precipitate an opportunity for task rescheduling, i.e., synchronizing a

scheduling event. However, reusable parts should not rely upon a delay of zero to effect task rescheduling given the current lack of uniformity among implementations of Ada runtime systems. An implementation may treat this abstraction as a null construct and continue execution of the enclosing part.

It has been recognized that a program enclosing reusable parts must exhibit functionally identical execution when transported among different execution environments⁷. A corollary of this would advise against relying upon the functionally equivalent execution guaranteed by the Ada standard when parts of a program are to be widely reused. However, even this desideratum cannot guarantee successful part reuse, particularly among real-time applications where there is a propensity to exploit language features. For example, in the instance where two parts are combined from different programs, the existence of implicit, but conflicting, runtime system dependencies would result in aberrant execution behavior.

While the prohibition of many implementation dependencies must be included in the programming discipline, relaxation of the prohibition may be essential in order to achieve predictable and reusable execution behavior. The relaxation would most likely occur where reusable paradigms are provided that stipulate implementation characteristics based upon formal analysis of the critical timing constraints.

Performance. The production of reusable software can be achieved only through more intellectually-intensive software development practices and by sacrificing some degree of performance efficiency. For real-time applications the tradeoff between reusability and performance efficiency is a dominant issue, especially in the presence of scheduling shared resources. The semantic elegance and versatility of the Ada tasking abstractions may impose both execution and storage penalties upon an application that are not competitive with those imposed by a compact rudimentary real-time executive kernel.

In many instances, variations in performance efficiency may depend upon the individual runtime system implementation. However, variations may result from the design and implementation strategies

selected for an application. Consequently, the programming discipline must provide sufficient guidance so that strict adherence to Ada reusability rubrics does not overwhelm the actual functional processing of a part. For example, a reusable part that implements a computational intensive algorithm required to perform frequent numeric conversions to accommodate different numeric types, will be reused only if the conversion execution costs are insignificant compared to the actual computations. In addition, the discipline should recognize the effects of compilation techniques that offer the opportunity for variations in performance efficiency. The performance efficiency of using dynamic arrays in unconstrained record types can differ significantly among implementations and typifies a common construction warranting attention and evaluation when used in a reusable part.

A Disciplined Approach

A discipline for real-time programming was advocated in a paper by Nicklaus Wirth⁶. The motivation for the discipline was to provide a straightforward approach toward analytically verifying the reliability of real-time programs. In order to support the discipline, the use of suitable abstractions for expressing concurrent program execution and synchronization was deemed essential to minimize dependencies upon processing speed. Through these abstractions logical assertions would allow the validity of a program to be deduced from the program's text with the same assurance as for a serially executed program. Such a discipline would increase the potential for software reuse.

The essence of the proposed discipline is that the reasoning and facilities for real-time programming should be limited extensions to those supporting multiprogramming which, in turn, should be limited extensions to those used for sequential programming. The important contribution of the discipline is managing the complexity of real-time constraints in the presence of processor sharing. The discipline requires that processor sharing be ignored in assertions of the part's computational state and confined to analyzable timing considerations of processor utilization.

Finally, the discipline should be decisively shaped by the programming

language. In this paper, the discipline is shaped by Ada, its runtime system, and tenets for software reuse.

Fundamentals.

The fundamentals of disciplined reusable Ada programming for real-time applications adapt the principles of Wirth's discipline to programming practices for achieving software reuse. Conflicts between expressing time-dependent execution behavior and writing reusable code must be reconciled. Reconciliation compromises part reusability to the extent that only limited reusability is achievable in the presence of critical hard timing constraints. At a minimum, the design for the part should be reusable if part reusability is claimed.

The formulation of binary and general semaphores has been used by Wirth to illustrate implicit time dependencies that may occur in the simplest of real-time applications. While the dangers of using these paradigms have been cited⁹ to argue for the safety of the Ada rendezvous, their use in real-time applications is frequently necessary for efficiency, since their analogues are commonly supported by the processing resource. Therefore, they provide a pedagogical example of how to apply some of the fundamentals of a discipline when adapted to Ada.

Recent work has indicated that even such classical paradigms are not without flaws¹⁰ unless reused under execution environments where the timing dependencies are well-defined. Therefore, wary of this admonition, a general semaphore is constructed from two binary semaphores by directly translating (reusing) a correct implementation of the paradigm¹¹ to Ada. The general semaphore is then transformed into a reusable Ada part which is critiqued with respect to its use for real-time applications. Finally, the difficulty of instrumenting the part for real-time use is presented.

Programming for Reuse.

General tenets for programming reusable Ada parts have been described in the literature¹². In addition, criteria for reusability have been defined that qualify a part as weakly, effectively, or strongly reusable⁷. Weakly reusable parts require extensive source modifications and have limited application; strongly

reusable parts require minimal source modifications and have widespread application. In this paper, programming effectively reusable parts is emphasized. These parts possess a high pragmatic potential for reuse.

Binary Semaphore. A binary semaphore to protect critical regions of code may be implemented in Ada using a trivial server task type idiom⁰. The following two variations provide for the binary semaphore to have either a "locked" or "unlocked" condition initially. The semaphore is constructed as an abstract type encapsulated in a simple package that provides an interface specification reflecting the accepted notation for binary semaphore operations. A task type is reused to achieve the different initial conditions.

```
package Binary_Semaphore_Package is
  type Locked_Binary_Semaphore_Type
    is limited private;
  type Unlocked_Binary_Semaphore_Type
    is limited private;
  procedure P (Sema : in Locked_Binary_Semaphore_Type);
  procedure V (Sema : in Locked_Binary_Semaphore_Type);
  procedure P (Sema : in
    Unlocked_Binary_Semaphore_Type);
  procedure V (Sema : in
    Unlocked_Binary_Semaphore_Type);
  pragma INLINE (P, V);
private
  task type Unlocked_Binary_Semaphore_Type is
    entry P V;
    entry V P;
  and Unlocked_Binary_Semaphore_Type;
  type Locked_Binary_Semaphore_Type
    is new Unlocked_Binary_Semaphore_Type;
end Binary_Semaphore_Package;
package body Binary_Semaphore_Package is
  task body Unlocked_Binary_Semaphore_Type is
    begin
      loop
        accept P V;
        accept V P;
      end loop;
    end Unlocked_Binary_Semaphore_Type;
  procedure P (Sema : in
    Locked_Binary_Semaphore_Type) is
    begin
      Sema.V P;
    end P;
  procedure V (Sema : in
    Locked_Binary_Semaphore_Type) is
    begin
      Sema.P V;
    end V;
  procedure P (Sema : in
    Unlocked_Binary_Semaphore_Type) is
    begin
      Sema.P V;
    end P;
  procedure V (Sema : in
    Unlocked_Binary_Semaphore_Type) is
    begin
      Sema.V P;
    end V;
end Binary_Semaphore_Package;
```

The above package illustrates, upon initial inspection, a reusable implementation for binary semaphores. However, it is likely that for real-time applications, performance considerations

would substantially reduce its utility unless the server task type idiom was appropriately optimized to minimize the normal overhead associated with a task entry call. In addition, this particular task idiom fails to provide a suitable method for self-termination of the task, thereby, requiring some external action, viz., an abort, to be employed. The idiom can be modified to enclose the accept statements within a select statement that includes a terminate alternative, but this would most likely increase overhead. Understanding these limitations the abstraction may be reused to construct a general, or integer, semaphore.

General Semaphore. A general semaphore is constructed as the following abstract type encapsulated in a generic package using the binary semaphore package. Again, the interface specification reflects the accepted notation for general semaphore operations.

```
with Binary_Semaphore_Package;
generic
  type Semaphore_Count_Type is range (0);
package General_Semaphore_Package_Template is
  type General_Semaphore_Type is limited private;
  procedure P (Sema : in out General_Semaphore_Type);
  procedure V (Sema : in out General_Semaphore_Type);
  Semaphore_Error, Invalid_Semaphore : exception;
private
  use Binary_Semaphore_Package;
  type General_Semaphore_Type is
    record
      Mutex : Unlocked_Binary_Semaphore_Type;
      Wait : Locked_Binary_Semaphore_Type;
      Count : Semaphore_Count_Type
        is Semaphore_Count_Type'Last;
    end record;
  and General_Semaphore_Package_Template;
package body General_Semaphore_Package_Template is
  procedure P (Sema : in out General_Semaphore_Type) is
    begin
      P (Sema.Mutex);
      Sema.Count := Sema.Count - 1;
      if Sema.Count < 0 then
        V (Sema.Mutex);
        P (Sema.Wait);
      end if;
      V (Sema.Mutex);
    exception
      when Constraint_Error =>
        V (Sema.Mutex); ...
        raise Semaphore_Error;
    end P;
  procedure V (Sema : in out General_Semaphore_Type) is
    begin
      P (Sema.Mutex);
      Sema.Count := Sema.Count + 1;
      if Sema.Count <= 0 then
        V (Sema.Wait);
      else
        V (Sema.Mutex);
      end if;
    exception
      when Constraint_Error =>
        V (Sema.Mutex); ...
        raise Semaphore_Error;
    end V;
  begin
    if 0 not in Semaphore_Count_Type then
      raise Invalid_Semaphore;
    end if;
  end General_Semaphore_Package_Template;
```

The above package is a reusable and safe implementation of the general semaphore. It provides a semaphore abstraction that may be used to create semaphores of differing capacities, i.e., each semaphore may be reserved by a different number of clients. The formal generic parameter, Semaphore_Count_Type, specifies limits for both the number of clients that may reserve the semaphore, Semaphore_Count_Type'Last, and for the number of clients that may be queued waiting to reserve the semaphore, Semaphore_Count_Type'First. Through the use of a general semaphore, access to a shared resource may be controlled by restricting the number of concurrent clients. When the semaphore has reached its specified capacity, subsequent clients are required to wait until a reservation is released. For example, in the following code fragments two semaphores are declared; one is equivalent to a binary semaphore and the other allows a maximum of three client parts to reserve the semaphore.

```
subtype Binary_Count_Subtype
  is Integer range -Max_Queue_Size..1;

package Binary_Semaphore_Package is new
  General_Semaphore_Package_Template
    (Semaphore_Count_Type =>
      Binary_Count_Subtype);
use Binary_Semaphore_Package;

Binary_Semaphore : General_Semaphore_Type;

subtype Quarternary_Count_Subtype
  is Integer range -Max_Queue_Size..3;

package Quarternary_Semaphore_Package is new
  General_Semaphore_Package_Template
    (Semaphore_Count_Type =>
      Quarternary_Count_Subtype);
use Quarternary_Semaphore_Package;

Quarternary_Semaphore : General_Semaphore_Type;
```

From a performance efficiency perspective the implementation may be unsuitable for several reasons. The most compelling reason is the necessity for two tasks to service the binary semaphores; even in the presence of task optimizations, the efficiency of the abstraction is likely to be compromised. Also, it is dubious that the ability to "inline" procedures will necessarily improve efficiency given that the subprogram bodies are nontrivial and rely upon exception handlers to control misuse of the abstraction.

Programming for Performance.

Programming practices to increase performance efficiency vary with a particular language. These practices

enable parts to be optimized for minimal execution time or economical utilization of storage. Understanding the relative efficiency of programming constructs is important. A construct that contributes to the generalization of a part may incur unacceptable execution time or storage penalties for real-time applications.

While a programming discipline must not eschew performance efficiency, striving for optimal performance of a part does not guarantee its successful reuse for real-time applications. Finally, a discipline must carefully delineate between programming optimizations and code optimizations when striving for performance efficiency. Code optimizations are likely to compromise part reuse and are inappropriate to a discipline.

Refined General Semaphore. A refined general semaphore that addresses some of the criticisms associated with an exact Ada transformation of the paradigm is achieved by improved utilization of the Ada tasking model. The semaphore is constructed using an abstract type encapsulated in a generic package using an equivalent interface specification.

```
generic
  type Semaphore_Count_Type is range (<);
package Ada_Semaphore_Package_Template is
  type General_Semaphore_Type is limited private;
  procedure P (Some : in General_Semaphore_Type);
  procedure V (Some : in General_Semaphore_Type);
  pragma INLINE (P, V);
  Semaphore_Error, Invalid_Semaphore : exception;
private
  task type General_Semaphore_Type is
    entry P;
    entry V;
  and General_Semaphore_Type;
end Ada_Semaphore_Package_Template;
package body Ada_Semaphore_Package_Template is
  task body General_Semaphore_Type is
    Count : Semaphore_Count_Type
      is Semaphore_Count_Type'Last;
  begin
    loop
      begin
        select
          when Count > 0 =>
            accept P do
              Count := Count - 1;
            end P;
          or
            accept V do
              begin
                Count := Count + 1;
              exception
                when Constraint_Error => ...
                  raise Semaphore_Error;
            end V;
          or
            terminate;
        end select;
      exception
        when Semaphore_Error =>
          null;
      end;
    end loop;
  end General_Semaphore_Type;
```

```

procedure P (Some : In General_Semaphore_Type) is
begin
  Some.P;
end P;

procedure V (Some : In General_Semaphore_Type) is
begin
  Some.V;
end V;

begin
  if 8 not in Semaphore_Count_Type then
    raise Invalid_Semaphore;
  end if;
end Ada_Semaphore_Package_Template;

```

The most important aspect of this refinement is the elimination of the task implementing the binary semaphore used to serialize access to the semaphore count. The synchronization guarantee of the rendezvous affords a straightforward alternative for protecting the logical consistency of the semaphore count. Consequently, performance efficiency with respect to execution and storage may be improved. Furthermore, the refinement appears to simplify reasoning about execution behavior particularly with respect to blocking conditions. Unfortunately, by using the rendezvous to serialize access to the semaphore count, the ability to limit the number of clients waiting for the semaphore has been relinquished. As a result, the procedure bodies of the semaphore operations are trivial, allowing them to be inlined. The triviality was achieved by promoting the exception handling for detecting semaphore misuse to a single accept body within the server task and allowing an exception to propagate to the client part. It should be appreciated that detecting semaphore misuse is only illustrative and is not included in the original paradigm. (One of the dangers of this paradigm is its lack of protection against misuse in a hostile environment, e.g., arbitrary V operations.)

If the accepted notation for semaphores is unnecessary, the syntactic camouflage of the procedural interface can be eliminated allowing the task entries to be referenced more efficiently. No practical loss in reusability results since a procedural interface may still be obtained by renaming statements for each semaphore.

Programming for Real-Time.

Adapting a reusable part for real-time applications presents a challenge. The paucity of formal techniques available to specify the intricacies of time-critical execution precludes the confident and rigorous application of a practical

discipline. Consequently, an informal approach that introduces a greater degree of predictability into the timing behavior of an executing reusable part must be developed.

Intuitive in the approach, in minimizing the interaction of the temporal semantics that may influence execution behavior. Simplifying these interactions by carefully restricting concurrency may often be sufficient to formulate logical assertions that predict execution behavior. This requires that the critical timing dependencies for a part be accurately interpreted in terms of a part's execution behavior. For example, a general semaphore should block the execution of a client part only if the semaphore cannot be reserved. If it is not possible to guarantee this, then the potential delay from any additional kind of blocking must be predictable for the part to be reused.

The eventual acceptance of a reusable part, such as the above generic package, for hard real-time applications will frequently depend upon its effect on the application's execution scheduling requirements. Normally these requirements mandate that the most urgent processing within the application be completed on time with a minimum of delay; namely, that execution is never blocked unless deliberately self-imposed, e.g., processing has been completed and must await some event. Urgency necessarily introduces the notion of priority. Priority in Ada is conveyed through the use of a pragma and is discerned as a potential impediment to part reuse. Consequently, only through disciplined programming derived from proven scheduling theory can the use of priority be justified.

Because formal scheduling theory for applications distributed among multicomputers remains an area of ongoing research, scheduling is commonly restricted to applications that execute on a single computer. Therefore, it is in the context of a single computer that a part's effect on scheduling is reviewed with respect to the programming discipline. This is not necessarily invalidated when the context is extended to include multicomputers.

Priority Inversion. A deleterious effect of the Ada tasking model on predictable scheduling execution behavior is the potential for priority inversion⁵.

Priority inversion refers to the condition where a lower priority task is executing and is preventing (blocking) the execution of a higher priority task. Each time a higher priority task calls an entry of a task that may have been called by a lower priority task, the opportunity for priority inversion exists. Consequently, a reusable part using the tasking model must avoid precipitating priority inversion in an application. In particular, rendezvous engagements require careful analysis to ensure that the first-in-first-out protocol queuing of entries can be neutralized. While paradigms using entry families can circumvent this protocol, the attendant increase in execution overhead may limit their reuse in real-time applications¹³. In addition, the inability for a part to change the priority assigned to a task prevents expediting the execution of blocking tasks.

Reexamining both versions of the general semaphore shows that there are no safeguards against the occurrence of priority inversion when caused by clients of different priorities. This is exacerbated by the fact that when a lower priority client is blocking a higher priority client, the code protected by the semaphore is executing at the lower priority, thereby reducing any beneficial effect from assigning the semaphore tasks a high priority. Furthermore, without analyzing the client tasks, the blocking time is unpredictable.

Regulating Task Interaction. An approach to minimizing priority inversion, consistent with Wirth's discipline, strictly regulates the interactions between client and server tasks. Wirth's discipline focused upon a specific variety of server task, viz., device drivers, based upon simple utilization analysis. More recent research¹⁴ provides evidence that when these interactions rigorously conform to analytic scheduling algorithms the blocking of high priority tasks for a simple class of client/server tasks can be successfully minimized within predictable bounds. The algorithms require that the overall timing constraints of an application are subject to the rate monotonic scheduling theorems, i.e., tasks with the shortest execution periods are given highest priority.

One result of this research, the priority ceiling protocol¹⁵, is particularly relevant to disciplined programming. The protocol provides the necessary understanding for continuing to

explore regulatory use of the task model coupled with explicitly defined runtime system implementation dependencies as a practical means for programming reusable parts for hard real-time applications. Therefore, it is discussed as a significant contribution toward disciplined programming of Ada tasks in the context of the previous semaphore examples. The discussion is incomplete and should not be interpreted as an authoritative treatment of the protocol. It is provided to illustrate that only through formal reasoning about critical timing dependencies can reliable abstractions and paradigms be devised for effective reuse.

Priority Ceiling Protocol. The priority ceiling protocol assumes the use of binary semaphores to synchronize access to shared resources or critical regions of code. It minimizes the time a higher priority task is blocked from reserving a semaphore by lower priority tasks. This time is bounded by the maximum time a lower priority task may reserve a semaphore. In addition, it guarantees avoidance of nontrivial forms of deadlock in the presence of multiple semaphores.

The implementation of this protocol requires two important conditions to be satisfied. The first condition is that when a task blocks the execution of higher priority tasks from reserving a semaphore, this task should execute at the highest priority of all tasks it currently blocks. The blocking task is said to inherit the priority of the highest blocked task. The second condition is that a task may reserve a semaphore only if it is executing at a higher priority than the highest priority, i.e., the ceiling priority, that may be inherited by the tasks it preempts. The two conditions are sufficient to achieve a prioritized ordering of tasks that minimizes the time a high priority task is blocked when attempting to reserve a semaphore.

It is clear from the Ada task model that these two conditions would not be achievable by the implementation of the binary semaphore presented earlier. The priority inheritance of the rendezvous is limited to that of the client, and the execution priority of the code protected by the semaphore is static (unless it is in the body of an accept statement). Consequently, different task idioms, that do not preclude meeting the two conditions, must be used for client/server interactions. These task idioms must

adhere to specific rules that reduce the nondeterminism of task execution. In addition, these idioms may assume an explicit dependency upon a "friendly", but valid, implementation of the Ada runtime system with respect to the execution freedom that may govern tasks having no specified priority.

The rules are briefly stated in terms of client and server tasks. A server is, in essence, a semaphore whose entries control access to critical regions, and a client is simply a non-server task that calls at least one server task.

1. Each non-server task must be assigned a priority consistent with rate monotonic theory.
2. All server tasks must be assigned either no priority or a priority higher than the highest priority non-server task.
3. Each server task must comprise a single continuous loop that encloses an unguarded select statement.
4. The select statement must enclose only one or more accept statements and a terminate alternative.
5. Nested accept statements must not be used.
6. Conditional and timed entry calls must not be used.

The application of these rules result in restricting an application to the following task idioms if priority inversion is to be controlled:

```
task type Server_Task_Type is
  entry Critical_Region_1 (...);
  ...
  entry Critical_Region_n (...);
end Server_Task_Type;
task body Server_Task_Type is
begin
  loop
    select
      accept Critical_Region_1 (...) do
        ...
      end Critical_Region_1; ...
    or
      accept Critical_Region_n (...) do
        ...
      end Critical_Region_n;
    or
      terminate;
    end select;
  end loop;
end Server_Task_Type;

task type NonServer_Task_Type is
  pragma PRIORITY (...);
end NonServer_Task_Type;
task body NonServer_Task_Type is
begin
  ... -- Non-server is client task.
  Server_Task_1.Critical_Region_1 (...); ...
end NonServer_Task_Type;
```

The use of these idioms in themselves do not guarantee that the above conditions are satisfied; they merely control task interaction so that blocking of non-server tasks becomes predictable when the conditions are satisfied. The efficacy of the idioms depends upon the Ada runtime system ensuring that the conditions are satisfied. The first condition is accomplished by requiring that before a client task is placed on an entry queue for a server task, it must be executing at a priority higher than the ceiling priority of any server executing directly or indirectly for another client. The requirement results in queuing of only one client for a server, thereby eliminating the effect of first-in-first-out queues in favor of a single prioritized queue of blocked tasks. The second condition is accomplished by allowing server tasks to be rescheduled as required by the set of tasks that are currently blocked. The rescheduling is legitimate only when no explicit priority is assigned to server tasks (rule 2). In this instance, the Ada standard does not prohibit the runtime system from scheduling a server task to effect priority inheritance.

Applying Ceiling Rules. Applying the rules of the protocol to the binary semaphore example, a reusable package is constructed as follows:

```
generic
  with procedure Critical_Region;
package Binary_Semaphore_Package_Template is
  type Binary_Semaphore_Type is limited private;
  procedure P_and_V (Some : in Binary_Semaphore_Type);
private
  task type Binary_Semaphore_Type is
    entry P_and_V;
    and Binary_Semaphore_Type;
  end Binary_Semaphore_Type;
package body Binary_Semaphore_Package_Template is
  task body Binary_Semaphore_Type is
    begin
      loop
        select
          accept P_and_V do
            Critical_Region;
            end P_and_V;
          or
            terminate;
          end select;
        end loop;
      end Binary_Semaphore_Type;
  procedure P_and_V (Some : in Binary_Semaphore_Type) is
    begin
      Some.P_and_V;
    end P_and_V;
  end Binary_Semaphore_Package_Template;
```

The functionality of the implementation is similar to the earlier version; however, the two traditional semaphore operations have been collapsed into a single operation that is explicitly

associated with the resource to be protected, i.e., the critical region. Consequently, programming of critical regions would differ. For example, the following programming scheme required by the original version:

```
task body Task_1 is
begin
  P (Sema);
  Critical_Region;
  V (Sema);
end Task_1;

task body Task_2 is
begin
  P (Sema);
  Critical_Region;
  V (Sema);
end Task_2;
```

would be changed to:

```
task body Task_1 is
begin
  P and V (Sema);
end Task_1;

task body Task_2 is
begin
  P and V (Sema);
end Task_2;
```

This reduces the freedom with which the semaphore can be used, viz., the clients cannot use the semaphore to protect arbitrary critical regions. It is apparent that allowing such freedom is not conducive to analyzing the predictability of task execution and, therefore, qualifies as a potential rule that would be included in a programming discipline.

The construction of a reusable package for the general semaphore becomes untractable using the priority ceiling server task idiom. The guard on the select alternative of the refined version contravenes rule 3 and must be removed, otherwise bounds upon blocking time are no longer predictable. Its removal invites reconsidering the original version of the general semaphore using two binary semaphores to protect the count reserving the critical region. Because the integrity of the count is not guaranteed outside of the server task implementing the binary semaphores, the call to the server task implementing the queuing semaphore must now be enclosed in the accept statements. The result of this revision precipitates self-imposed deadlock that cannot be precluded by the protocol.

Only by increasing task interaction is it possible to imitate the functionality of the general semaphore using the prescribed task idioms. This clearly reaffirms that the general semaphore is an abstraction oriented toward time-independent concurrent execution, whereas the priority ceiling idiom emphasizes predictable synchronized execution sharing a single processing resource.

Specification for a Discipline

The disciplined approach discussed in the preceding section demonstrates some difficulties of developing reusable parts for real-time applications. Adapting a reusable part to satisfy hard timing constraints may not be possible. Conversely, the reuse of a part from a real-time application may be effective only when reused in applications that suffer the same timing constraints.

The specification of a programming discipline that ameliorates the difficulties is essential. The evolution of such a discipline cannot progress unless there is an underlying basis of practice and theory. Today no such basis exists since real-time and reusability technology are distinct crafts within the programming community. To establish the basis for a discipline, a gradual and systematic transition of these technologies into programming practices is necessary. This becomes practical only through formal guidelines, proven paradigms, and uniformity criteria for Ada runtime systems that sustain the guidelines and paradigms.

Guidelines.

An example of informal guidelines that are consistent with an approach to disciplined programming has been promulgated in an Ada reusability handbook¹⁰. These guidelines include the fundamentals for developing reusable parts. While the guidelines do not address developing parts that are subject to hard timing constraints, many contribute to an awareness of the temporal implications that compromise part reuse.

Developing and using these guidelines indicates that it is unlikely that empirical guidelines can be formulated to respect hard timing constraints. However, specified guidelines can result in programming parts where reusability has been moderated with regard to performance efficiency and execution criticality. For example, a guideline restricting a sub-program supplied as an actual parameter to a generic instantiation from use outside of that generic unit would be applicable in the context of the generic binary semaphore package.

Paradigms.

The derivation of paradigms that can be reused within real-time applications is essential to advancing the discipline.

Paradigms should synthesize related guidelines to comply with formalized timing constraints. Without the paradigm, the individual guidelines might possibly appear counter-intuitive. The value of such paradigms is evident from the discussion on priority inversion. In this instance the server task idiom is the derived paradigm.

The discipline should include the reuse of meta-paradigms to support real-time applications. These paradigms are used in the construction of paradigms that comply with formalized timing constraints. Alternatively, they may support the use of a specific guideline. An example of a meta-paradigm would be a generic package that ensured a subprogram implementing a critical region is reused only within a specific context. An outline for a meta-paradigm to accomplish this reusing the binary semaphore package is as follows:

```
package Critical_Region_Package is
  type Region_Guard_Type is limited private;
  procedure A_Critical_Region
    (Region_Guard : in Region_Guard_Type);
  generic
    with procedure Critical_Region
      (Region_Guard : in Region_Guard_Type);
  package Binary_Semaphore_Package_Template is
    ...
  end Binary_Semaphore_Package_Template;
  Unsafe_Use : exception;
private
  function Raise_Unsafe_Use return Boolean;
  type Region_Type is (Non_Critical, Critical);
  type Guard_Type
    (Guard : Region_Type := Non_Critical) is
    record
      case Guard is
        when Critical =>
          null;
        when Non_Critical =>
          Unsafe : Boolean := Raise_Unsafe_Use;
      end case;
    end record;
  type Region_Guard_Type is
    record
      Region_Guard : Guard_Type;
    end record;
end Critical_Region_Package;
package body Critical_Region_Package is
  function Raise_Unsafe_Use return Boolean is
    Safe : Boolean := False;
  begin
    if Safe then
      return True;
    else
      raise Unsafe_Use;
    end if;
  end Raise_Unsafe_Use;
  procedure A_Critical_Region ...;
  package body Binary_Semaphore_Package_Template is
    task body Binary_Semaphore_Type is
      begin
        loop
          select
            accept P_and_V do
              Critical_Region
                (Region_Guard_Type'(Region_Guard =>
                  Guard_Type'(Guard => Critical)));
            end P_and_V;
          ...
        end Binary_Semaphore_Package;
      end Critical_Region_Package;
```

The above paradigm requires that each critical region include an additional parameter to restrict its reuse to the binary semaphore package. This necessitated a minor change to the binary semaphore package.

In the near term, the use of paradigms may be the only practical means for developing partitioning strategies for real-time applications that must be distributed among multiple computers. Paradigms such as those derived from virtual nodes¹⁷ place specific restrictions upon the locality of parts, thereby extending the programming discipline into more global concerns that have been left unaddressed in this approach to disciplined programming.

Uniformity.

A prevailing maxim when programming Ada reusable parts is to avoid dependencies on runtime system implementations. For part reuse in real-time applications the strict adherence to this maxim must be relaxed because of application timing constraints. Critical timing constraints can be achieved only when runtime system implementations conform to precisely defined semantic interpretations of the Ada standard that lead to practical real-time execution models.

The priority ceiling protocol is a convincing example for programming reusable parts based upon a "friendly" implementation of the runtime system. The example demonstrates that its parts are to be reused in a real-time application, where priority inversion must be bounded, dependencies upon the runtime system can be used to program effectively reusable parts. This warrants a programming discipline that, while legislating against uncontrolled use of implementation dependencies, is sufficiently flexible to promulgate controlled use of Ada real-time models. Controlled use implies that uniformity criteria be established for runtime systems. The discipline would require that the reusability of a part be qualified with respect to the uniformity criteria satisfied by a runtime system. For example, a part that must tolerate a Storage_Error exception might require that the runtime system guarantee the execution of the exception handler for the frame that suffered the exception. Consequently, reuse of the part would be qualified by this requirement.

A corollary is that all reusable parts constituting an application must depend upon the same or compatible uniformity criteria. This would ameliorate the potential problem that currently exists when parts with conflicting dependencies are combined. Conflicting dependencies become detectable since the discipline would require the assertion of uniformity criteria for each reusable part.

Leveraging the Ada runtime system to program reusable parts for real-time applications is expected to increase. Initiatives such as the International Standards Organization's Uniformity Rapporteur Group are evidence that uniformity of runtime systems will evolve in time to support the objective of disciplined programming.

Conclusions

Ada promotes many of the programming language practices that Wirth perceived when proposing a discipline for real-time programming. This paper has borrowed from this proposal to argue the necessity for extending the discipline to construct reusable Ada parts for real-time applications. In addition, it has emphasized that the desiderata for a discipline comprise guidelines and paradigms that rely on sound theoretical principles and improved uniformity of Ada runtime systems.

It is apparent that extending the discipline to increase software reuse among real-time applications offers a challenge not only to the fundamental principles of the original discipline but to Ada. This paper has provided a perspective on the scope of this challenge by suggesting a proposed discipline.

Acknowledgement

This work was supported in part by the U.S. Army Communications and Electronics Command (CECOM) Center for Software Engineering (CSE) under contract number DAAB07-85-C-K524, Task 7087-34040. The authors are pleased to acknowledge the contributions to real-time and reuse issues gained from attending the Real-Time Technical Interchange Meetings sponsored by the U.S. Army CECOM.

References

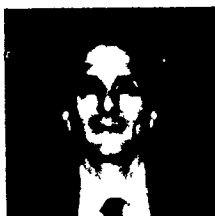
- [1] McDonnell Douglas Astronautics Co.: Common Ada Missile Package (CAMP), Tech. Report AFATL-TR-85-93 (May 1986).
- [2] Tracz, W.: Ada Reusability Efforts - A Survey of the State of the Practice, Proceedings of the Fifth Annual Joint Conference on Ada Technology and Washington Ada Symposium (1987).
- [3] Gargaro, A. and Pappas, T.: Reusability Issues and Ada, IEEE Software Vol. 4, No. 4 (July 1987).
- [4] U.S. Army CECOM/CSE: Analysis of the Impact of the Ada RunTime Environment on Software Reuse, Interim Report (September 1988).
- [5] Barnes, J.G.P. et al: Proceedings of the International Workshop on Real-Time Ada Issues, ACM SIGAda Ada Letters VII, 6 (Fall 1987).
- [6] ACM SIGAda Ada RunTime Environment Working Group: Catalog of Interface Features and Options (December 1987).
- [7] Computer Sciences Corporation: Ada Reusability Study, Tech. Report SP-IRD 9 (August 1986).
- [8] Wirth, N.: Toward a Discipline of Real-Time Programming, Comm. ACM 20, 8 (August 1977), 577-583.
- [9] Ichbiah, J. D. et al: Rationale for the Design of the Ada Programming Language, Honeywell Systems Research Center and Alsys, Inc., (February 1980).
- [10] Kotulski, L.: Comments on Implementation of P and V Primitives with Help of Binary Semaphores, ACM SIGOPS Operating Systems Reviews 22, 2 (April 1988).
- [11] Hemmendinger, D.: A Correct Implementation of General Semaphores, ACM SIGOPS Operating Systems Reviews 2, 3 (July 1988).

- [12] St. Dennis, R. J.: Reusable Ada Software Guidelines, Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences (1987).
- [13] U.S. Army CECOM/CSE: Real-Time Requirements Annex to the Ada Reusability Handbook, Interim Report (September 1988).
- [14] Software Engineering Institute: An Overview of Real Time Scheduling Algorithms, (June 1988).
- [15] Goodenough, J. and Sha, L.: Priority Ceiling Protocol - A Method for Minimizing the Blocking of High Priority Ada Tasks, Proceedings of the International Workshop on Real-Time Ada Issues, ACM SIGAda Ada Letters VIII, 7 (Fall 1988).
- [16] Computer Sciences Corporation: Ada Reusability Handbook, Tech. Report SP-IRD 11 (December 1987).
- [17] Hutcheon, A. D. and Wellings, A. J.: The Virtual Node Approach to Designing Distributed Ada Programs, Ada User Vol. 10, No. 1 (1989).



Frank Arico is an Ada specialist with Computer Sciences Corporation. He has been a principal contributor to several Ada projects, including one directed toward the evaluation of Ada reusability technology.

Mr. Arico received a MS degree in computer science from Temple University. He is a member of the ACM, SIGAda, and the IEEE computer society.



Anthony Gargaro is a lead scientist specializing in the use of Ada for defense systems. During his tenure at Computer Sciences Corporation he has served terms as Chairperson of the ACM Special Interest Group on Ada and as a member of the Federal Advisory Committee for Ada. In 1983 he was awarded the DoD Distinguished Service Award for contributions to the Ada Program.

Mr. Gargaro graduated in Numerical Analysis and Statistics from Brunel College, U.K. He holds the ICCP CDP and CCP, and is a member of the ACM SIG Board and the British Computer Society.

The Morehouse Object-Oriented Reuse Library System

Arthur M. Jones, Ph.D
Robert E. Bozeman, Ph.D
William McIver

Morehouse College
P.O. Box 131
Atlanta, Georgia 30314

Abstract

As a participant in the Ada House and Metrics Research effort being conducted for the Army Institute for Research in Management, Information, Communications and Computer Science (AIMICS), the Morehouse College Software Group has completed two tasks: A study of the appropriateness of several database models for supporting reusability tools in current and future integrated programming environments; the development of a conceptual model of an extensible integrated software development environment which utilizes a common object-oriented database subsystem. The Object-oriented Database model has been selected as the preferred model. Morehouse College is currently implementing a Ada reuse library system using existing object-oriented database technology.

1 LIBRARY MANAGEMENT ISSUES

1.1 Introduction

The idea of building software systems from existing components is very old, perhaps old enough to have been recognized by Sabbage. As a result of recent research, however, the concept of reusability has evolved in scope to include the knowledge gained during a software development life cycle as well as the software components themselves. That is, current opinion emphasizes that reuse archives should capture "process" along with "product".

Some researchers have predicted that the potential dividends from the investment in development and know-how is much greater than that from the software components. This, in spite of enormous technical barriers to the implementation of a practical reuse repository, was the compelling

motivation for this research.

This study of data models, library strategies, and engineering environments was principally influenced by the desire to accommodate information pertinent to software parts and the process by which they were developed.

The coupling of library management with configuration management reflects an assumption that the utility of a large-scale reuse library implementation, which captures and archives engineering experience as well as software components, must rely heavily upon the application of sound principles of configuration management.

This framework is a generalized organization scheme for reuse libraries. It provides a framework from which many types of libraries - large or small, public or private - may be described.

1.2 Library Attributes

The contents and use of one library may vary greatly with that of another. One library might be a large, national repository of part descriptions, for general, public access. Another might be a large, yet local company repository consisting of both parts and their descriptions for use in a proprietary project. While the latter library is private, any number of parts may have been imported from other libraries. Thus, a high degree of commonality among libraries should be sought.

As seen above, reuse libraries may be classified in terms of a number of attributes. Several major attributes are enumerated below.

1.2.1 Size -

The Size attribute represents the number of parts contained in a library. The purpose of this attribute is to provide the user with information which would help determine a search strategy for using the library. This attribute bears directly upon the manner and speed with which a user may navigate through the library. For example, a user might not opt for a "shotgun" search strategy when accessing a library which contains a very large number of parts.

1.2.2 Parts Domain -

The part domain attribute is a list of the classes of

software parts contained within the library (e.g. graphics, communications, etc.). Libraries may be single or multiple domain repositories. A Domain classification list would be stored with each library and should be derived from a standard set of classifications (Figures 1 and 3).

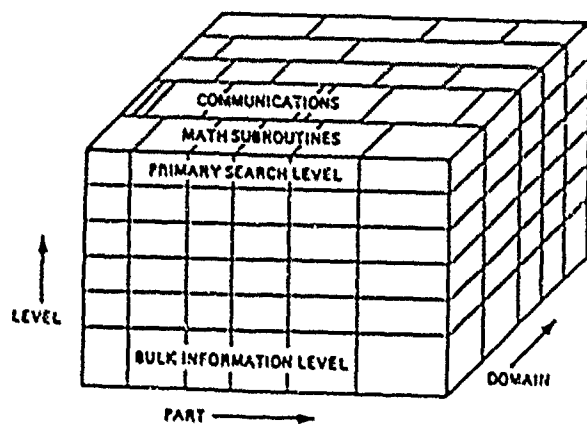


Figure 1. Library Structure.

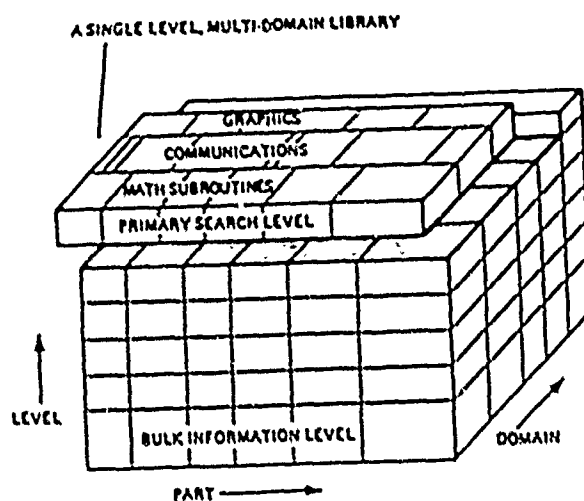


Figure 3. A single-level, multi-domain library.

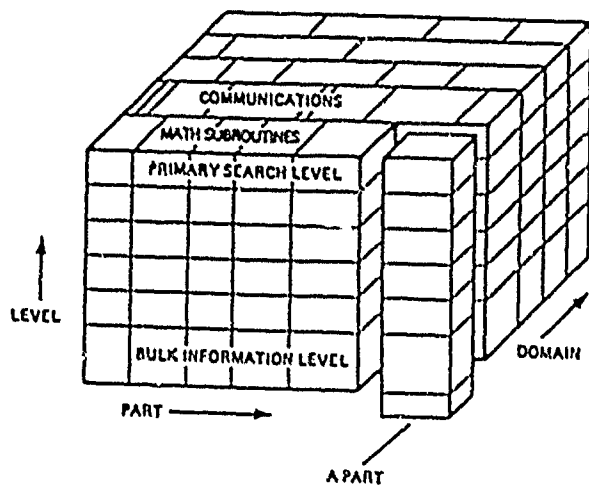


Figure 2. Part Structure.

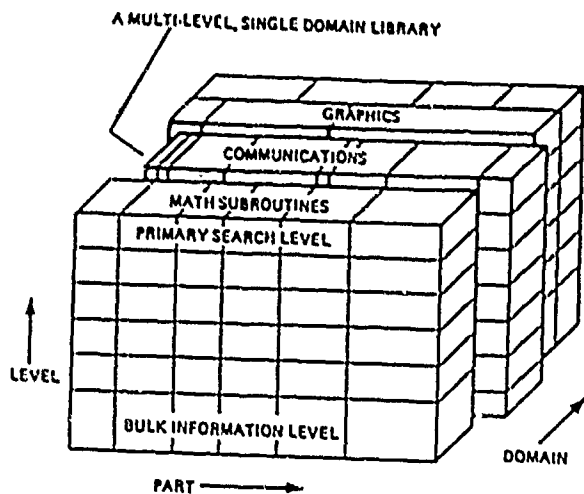


Figure 4. A multi-level, single-domain library.

1.2.3 Completeness -

Completeness is the extent of coverage of information that parts in a particular library contain. This can be expressed in terms of the presence or absence of information in each of the parts' information categories. Project libraries would probably tend to catalogue the most complete parts, while the most widely accessed libraries might only contain part information sufficient for conducting initial searches. We use the term LEVEL to refer to the rank of an information category within a part. In general, search information categories rank high while bulk information items rank low (Figures 3, and 5).

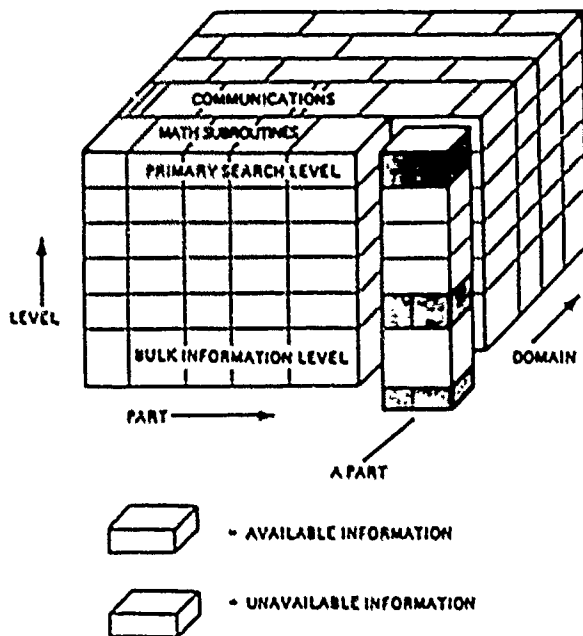
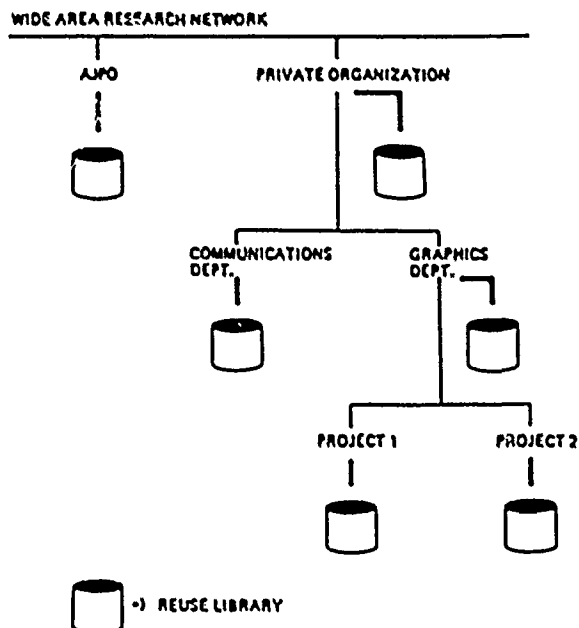


Figure 5. Part components.

1.2.4 Scope -

The Scope attribute describes the range of users which can ask the library to provide services. Scope supports the notion of libraries which are globally public or private, as well as that of libraries which are public or private within a certain level of a network domain. For instance, an organization may possess libraries which it deems public only within the organization. It may also declare other libraries to be private to certain groups within the

organization (Figure 6).



1.3 Library Parts

We define a *part* to be the fundamental, catalogued unit of a library. A *part* contains some or all of the products created during the software development life cycle phases of a software component. In addition, a *part* may have attributes stored with it along with its relationships to other parts [23].

The Ada Reusability Guidebook suggests a comprehensive list

of information categories and associated data elements that should be stored with every component upon inclusion in a library [23]. A partial listing is as follows:

Category

PART DESCRIPTION

Title
Type of Part (code, design, etc.)
Type of Function
Purpose of Function
Interface Requirements

PART CONSTITUENTS

Abstract
Requirement Specification
Functional Specification
Design
Algorithm or Function
Source Code
Object Code
Test Specification
Test Code
Test Data/ Results

PART HISTORY

Reason for Part Development
Date of Completion
Descriptions of Applications Used
Frequency of Use
Description of Development standards
Version Number

SUBMITTER DATA

Name
Address/ Network Address
Phone
Contact

PART ATTRIBUTES

Keywords(to search on)
Development Language
Host Environment (Computer and Operating System)
Target Environment (Computer and Operating System)

RESTRICTIONS

Government

Developer
Environment imposed, e.g. compiler, tools, peripherals,..
Reusability Metrics

Other categories of information include Disclaimers, Software support, Miscellaneous instructions, and Media.

1.3.1 Part Extensibility -

Initially, it is likely that only a minimum set of information categories can be agreed upon. However, a part is a "snapshot" of a continuing software development process and its products at some phase in the life cycle of a software component. The data items within a part will have to be updated when modifications to a software component, such as bug fixes, are made. Future software development tools and processes may necessitate the addition of new information categories to the library schema, as well as modifications to the existing schema. The library architecture therefore, provides for the addition and modification of information categories and their associated data elements.

1.3.2 Bulk Information -

Bulk information such as source code and documentation may be referenced by pointers for if storage permits, bulk information may be stored in a library along with the other information contained in a part.

It has been recognized that higher degrees of reuse of information contained in monolithic items, such as specification documentation, could be promoted by expressing the information in small, reusable groups. For instance, specification documents could be manipulated as sets of smaller specifications [3,5,8].

It is highly desirable, therefore, for the library management system to provide services for manipulating bulk information items in the appropriate logical parts. This could be accomplished by storing logical and behavioral qualities, such as a parsing mechanism and the appropriate syntax, of a bulk information item in the library. This has been demonstrated using object-oriented database systems [9,12].

2 THE REUSE LIBRARY SYSTEM

In this section we give an overview of our design.

Our design has three levels: an object-oriented database which manages a reuse libraries, an application interface to the object-oriented database, and an integrated software engineering environment. At the time of this writing, our work is being concentrated on the database level of the system.

2.1 The Integrated Software Engineering Environment Level

Our integrated software engineering environment will be constructed by coupling several existing tools in our research environment. Ada will be used to manage the subprocesses needed to execute the tools in the environment. In addition, the user interface will be constructed in Ada.

The motivations behind the use of integrated software engineering environments are well known. An environment consisting of a set of tightly coupled tools which presents a consistent user interface can stimulate system conception at a high-level of abstraction and can be configured to support a particular design methodology (e.g., object-oriented, procedural, etc.) [4,9]. The tools cooperate to support the development effort. A consistent user interface eliminates the need for the developer to perform mental context switches, which can reduce productivity [4,11].

In addition to having a common user interface among tools, a tightly coupled environment is one in which the tools share intermediate representations of software being developed in the environment. One approach that has been taken is to have the tools share a common database which manages all the forms and versions a software system may have during its life cycle [5,6,9]. The desirable characteristics of an integrated software engineering environment are:

1. that it be extensible, that is, it supports the convenient addition of tools to the environment;
2. that it supports interoperability, that is, all tools share the same underlying database representations, and they properly interpret these representations.

Extensibility can be achieved in a large part through the use of a common database [9]. Interoperability can be achieved through the selection of an underlying database model which has flexible and robust modeling capabilities (Figure 7) [5,6,9].

AN INTEGRATED SOFTWARE ENGINEERING ENVIRONMENT

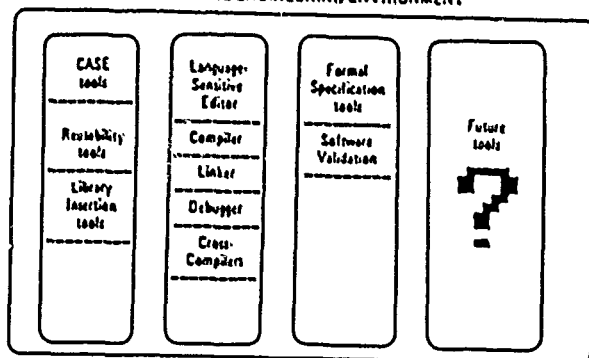


Figure 7. An integrated software engineering environment.

2.2 The Database Application Interface Level

The database application interface will provide a common protocol with which the tools in the integrated software environment will use to communicate with the database. The interface will also provide a bridge between the procedural features in Ada and the protocols used to communicate with objects in the database (Figure 8, 9).

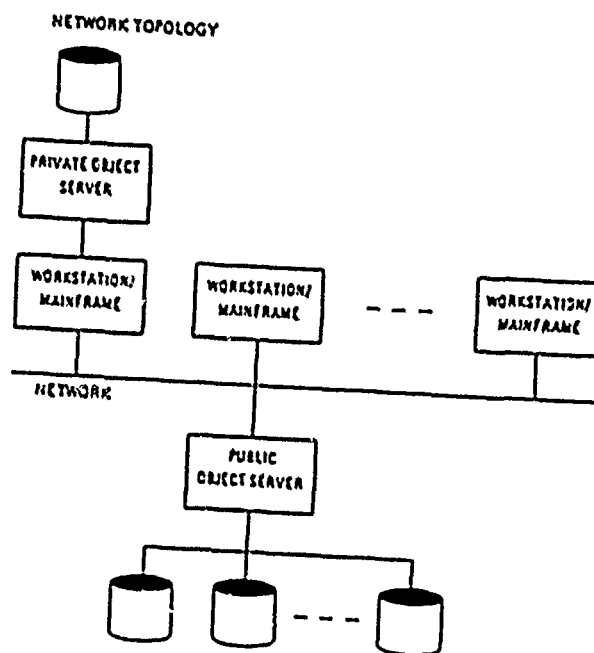


Figure 8. Network topology

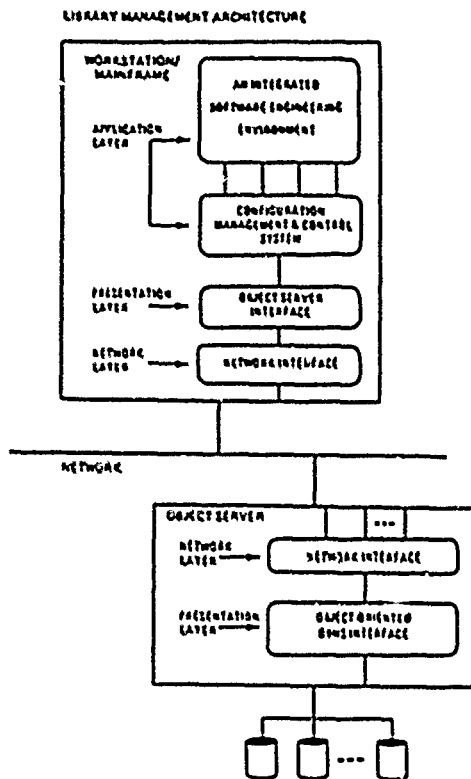


Figure 9. Library management architecture.

2.3 The Object Database Level

The database will support an extensible object schema for maintaining libraries of reusable software components. Most of the entities in the database will be complex persistent objects. These objects will be of several classes: library objects, part objects, several part constituent object classes, and several other special object classes.

Each object has both structural and behavioral properties. The structure of an object may be described in terms of the quantity and classes of constituent objects it contains. The behavioral properties of an object are described in terms of the methods (procedures) which may be used to communicate with the object to obtain services. Each class of objects will have methods defined to provide useful services to the application interface level, such as creating, displaying and modifying objects.

This system is being implemented using the Gemstone object-oriented database system [10].

2.3.1 Library Objects -

At the highest level of the database schema are library objects. Rather than treating the database server itself as a library, the schema defines library objects which are complex objects containing part objects. By defining a library as an object, a single database server can logically manage more than one grouping or library of parts. For example, an organization may wish to maintain separate libraries for each of its departments in a single database server (Figure 10, 11).

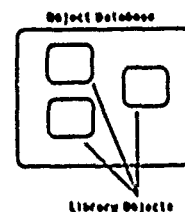


Figure 10. Object base structure.

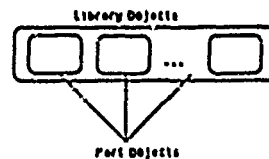


Figure 11. Library object structure.

2.3.2 Part Objects And Part Constituent Objects -

A part object is an extensible class of objects which may contain some or all of the products generated during the lifecycle of a software component. For example, a part object may contain some or all of the following information: part classifications, documentation, source code, test results, and part history. Each of these different types of information are represented by a corresponding class of objects whose instances are constituents of part objects (Figure 12).

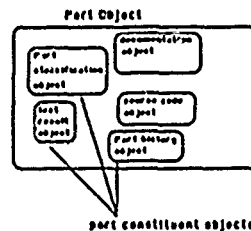


Figure 12. Part object structure.

3 SUMMARY

The immediacy of demands for a solution to the reusability problem poses a dilemma. While practical implementations must be sought now, a parallel effort must be mounted to develop reusability methodologies and tools that will be sufficiently flexible and robust to allow the incorporation of future programming technologies.

The object-oriented data model appears to have overwhelming advantages in comparison with competing ones, such as the

relational model. Notwithstanding the presence of mature, commercially-available relational databases, the extra burden of building library and configuration management subsystems in pursuit of an object-oriented approach should be borne.

References

1. Banerjee, J., Chou, Meng-Tai, Garza, J.F., Kim, M., Moorh, D., Sallou, M., and Kim, Myoung-Jon, Data Model Issues for Object-Oriented applications, ACM transactions on Office Information systems, January 1987, 51-66.
2. Burton, B.A., Aragon, R.W., Bailey, S.A., Koehler, R.D., and Hayes, L.A., The Reusable Software Library, IEEE Software, July 1987, 4125-32.
3. Burton, B., and Broids, M., Development of an Ada Package Library, Proceedings of the Fourth Annual Conference on Ada Technology, March 1986, 42-50.
4. Delsale, N., Henicasy, D., Schwartz, M., Viewing a Programming Environment as a Single tool, ACM, August 1987, 49-56.
5. Hudson, S.E. and King, R., Object-oriented database support for software environments, Proc of ACM SIGMOD Intl. Conference of Management of Data, May 1987.
6. Hudson, S.E. and King, R. The Cactus Project: Database Support for Software Engineering, IEEE Trans. on Software Engineering, June 1988.
7. King, Roger, An Object-Oriented/Semantic Approach to Ada Reusability and Life-Cycle Management, AIRMICS IFA, March 1988.
8. Leblanc, R. and Ornburn, S., "Revised Position Paper: Generic Specifications", Georgia Institute of Technology, Atlanta, July 1987.
9. O'Brien, Patrick, Halbert, D.G., Kilian, M.F., The Teellis Programming Environment, OOPSLA 1987 Proceedings, October 1987, 91-102.
10. Purdy, Alan, Schuchardt, B., and Maier, D., Integrating an Object Server with Other Worlds, ACM Trans. on Office Info. syst., January 1987, 27-47.
11. Reps, Thomas, Teitelbaum, Tim and Demers, Alan, Incremental Context-Dependent Analysis for Language-Based Editors, ACM Transactions on Programming Languages and Systems, July 1983.
12. Smith, Karen, and Zdonik, Stanley B., Intermedia: a Case Study of the Differences Between Relational and Object-Oriented Database systems, OOPSLA 1987 Proceedings, October 1987, 452-465.
13. Teitelbaum, Tim and Reps, Thomas The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, ACM, September 1981:563-573.
14. Wald, E., Reusability Guidebook, Software Technology for Adaptable, Reliable Systems Program, Department of Defense, 1986.

4 ABOUT THE AUTHORS

Arthur M. Jones received the B.A. degree in Mathematics from Morehouse College, Atlanta, Georgia and the Ph.D. degree in Mathematical Statistics from the University of Iowa, Iowa City, Iowa. He was associated with CMA in Chicago, Illinois as an actuary from 1961 to 1969. He has taught mathematics and computer science at both Morehouse College and Atlanta University, Atlanta, Georgia since 1975. Since 1982 he has served as director of the Morehouse Software Group. He was the principal organizer of the first "Annual National Conference on Ada Technology", which was held on the Morehouse College campus in January, 1983. His research interests include random spacings, goodness-to-fit tests, software engineering, computer-based instruction, and database methodologies.

Robert E. Bozeman is Professor of Mathematics at Morehouse College, Atlanta, Georgia and has taught there since 1973. He received the B.S. degree from Alabama A & M University in 1968 and the Ph.D. degree in Mathematics from Vanderbilt University in 1973. In 1981 he was part of a team in the Atlanta University Center involved in research on the Ada programming language. His primary emphasis was on the development of a library of the elementary mathematical functions in Ada. His research interests include orbital mechanics, numerical methods for ordinary differential equations and software engineering.

William McIver received the B.A. degree in Computer Science from Morehouse College, Atlanta, Georgia in 1986 and the M.S. degree in Computer Science from the Georgia Institute of Technology, Atlanta, Georgia in 1988. He is currently a research scientist with the Morehouse Software Group and instructor of Computer Science. In 1988, he was involved in the Ada Reuse and Metrics research effort being conducted for the Army Institute for Research in Management, Information, Communications and Computer Science. His research interests include object-oriented databases, object-oriented design, and human factors.



Arthur M. Jones

Robert E. Bozeman



William McIver

REUSE AND THE SOFTWARE LIFE CYCLE

Dany S. Guindi
W. Michael McCracken
Spencer Rugaber

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia

SUMMARY

Although some aspects of the software development life cycle have been adapted to deal with reuse, others have been ignored. In particular, the verification and validation (V&V) process has not been adapted to deal with reusable components. During the past year the authors have investigated the verification and validation of reusable software components. As part of that project, a review of different software development life cycle models was conducted in order to determine their applicability to the development of reusable software and the preferred V&V techniques associated with each. Among those studied were the "Waterfall," "2167," "2167A," the Software Reusability Guidebook, and the Spiral Model. In so doing, each stage of the different life cycles was studied from the perspective of the relevance of the prescribed methods to reuse and to V&V. This paper summarizes the results of that study and proposes some methodological changes geared towards making software development with Ada produce more reliable and reusable code.

Introduction

The ability to reuse software has been heralded as one of Ada's biggest benefits. Nonetheless, there have been few meaningful examples of software reusability within the Ada community.^{1,2} Part of the blame can be attributed to the fact that Ada is new and still not completely understood, that Ada compiler technology has only been in place for a few years, and that programming environments for Ada are still being developed. More significant, however, is the fact that the methods being used are those derived from traditional software development. Although some aspects of the software development life cycle have been adapted to deal with reuse, others have been ignored. In particular, the concepts of validation and verification (V&V) have not been adapted to deal with reusable components.

During the past year the authors have been involved in the development of software tools and methods to be used for the validation and verification of reusable software components.³ As part of that project, a review of different software development life cycle models was conducted in order to determine their effectiveness in the actual development of reusable software and the preferred V&V techniques associated with each. Among those studied were the "Waterfall," the "Spiral" model, "2167," "2167a," the "Domain-Oriented" model, and the Software Reusability Guidebook. This paper summarizes the results of the study and proposes some methodological changes oriented towards making software development with Ada produce more reliable and reusable code.

The Waterfall Model

The waterfall model⁴ consists of six stages: requirements, specifications, detailed design, coding and unit testing, integration and system testing, and maintenance. Each stage incorporates V&V activities. How to conduct V&V is not explicitly stated in the methodology and is thus up to the software developer. The model also does not explicitly indicate a preferred programming methodology, thus allowing the possibility of reusing parts by basically designing only to the level where the parts would fit.

The Waterfall Model was developed before reuse was a major concern. As such, it makes no comments on development for the sake of reuse. It is important, however, because it is the development method generally taught in computer science programs and because it is the standard against which other life cycle models are measured.

The Spiral Model

The Spiral Model was developed by Barry Boehm as a mechanism for reducing the risks involved in developing software using traditional methods. At each stage of the process, risk analysis is conducted and the developer follows the path which minimizes risk. The Spiral Model consists of several cycles, each cycle consisting of four phases. The first phase is used to elaborate the objectives and constraints of the current cycle. The second phase evaluates the alternatives with respect to the objectives and constraints expressed in the first phase of the cycle. The third phase consists of the development and testing of the specific product being developed. The methodology used during this

phase is up to the developers. The last phase is used to review the achievements of the current cycle and to initiate planning of the following cycles.

The model described above does not depend upon any particular development technology. It is simply an approach to managing software development. One of the objectives stated in the first phase of a cycle might be to achieve a certain level of V&V, the model itself does not describe how that would be done. Determining if such an objective is achievable is one of the problems that the developers would face in the second stage. The model as described allows for the incorporation of any available technology. Thus, if certain technologies became available for use within a project, (i. e. a system of reusable parts with an integration mechanism, a V&V tool, etc.), the model allows its incorporation.

DoD-STD-2167

DoD-STD-2167 establishes the requirements to be applied during the development and acquisition of mission-critical computer system software. It consists of a specific set of steps to be followed through the software life cycle. The system development cycle consists of four stages: the concept exploration stage, the demonstration and validation stage, the full-scale development stage and the production and deployment stage. Relevant software development usually occurs during the full-scale development stage. The software development cycle consists of six phases: 1) The software requirements analysis phase, 2) the preliminary design phase, 3) the detailed design phase, 4) the coding and unit testing phase, 5) the computer software component (CSC) integration and testing phase, and 6) the CSCI testing phase.

The software development life cycle proposed by 2167 includes formal and informal reviews and the development of formal and informal test procedures for V&V of the product. The specific design and testing methodologies used are left up to the user, although a top-down approach is suggested for the design, coding, integration and testing.

DoD-STD-2167 does discuss reuse. It encourages contractors to "incorporate into the current software design commercially available software, Government furnished software, and reusable software developed for other applications". However, it does point out the V&V of the reusable parts is the responsibility of the contractor.

DoD-STD-2167A

DoD-STD-2167A is a recent revision to DoD-STD-2167 that describes the

uniform set of requirements for software development which are applicable throughout the life cycle. It is important to note that DoD-STD-2167A does not describe a preferred life cycle nor does it impose a software development method. DoD-STD-2167A uses the same software life cycle described by DoD-STD-2167 as a sample to explain the set of requirements imposed by the standard. DoD-STD-2167A differs from DoD-STD-2167 in the level of details. Whereas DoD-STD-2167 tried to be very specific as to how certain activities are to be conducted, DoD-STD-2167A still expresses the need for such activities but does not go into the amount of detail as to how they are to be performed.

DoD-STD-2167A introduces a few new terms to define old ideas. One of them is *Non-developmental Software (NDS)*, which is any software that is not developed under the contract but is provided by the contractor, the Government or a third party. NDS can be considered to be equivalent to reusable software. DoD-STD-2167A encourages the use of NDS by requesting that the contractor consider incorporating NDS into the deliverable software and document its plans for using NDS. It gives permission to use NDS without approval from the contracting agency as long as the NDS is fully documented in accordance with the standard.

A final item of interest concerning DoD-STD-2167A is the continuous encouragement for the use of automated tools to support the software development effort. Such tools range from software engineering environments (SEEs), software test environments (simulation software, code analyzers, etc), to simple revision control systems. Whereas a great deal of detail has been omitted from DoD-STD-2167A, the document does touch on a few new areas and does encourage the use of automated tools that will hopefully fill in the amount of detail that is needed to perform the activities adequately.

The Domain-Oriented Software Life Cycle

A literature search did reveal one paper describing a software life cycle model that specifically addressed reuse. This was the Domain-Oriented Software Life Cycle model developed by Mark Simos at Unisys Corporation.¹⁰ The domain-oriented life cycle, on the other hand, closely models the way software is actually developed. It takes both a top-

down, "problem-driven" approach and a bottom-up, "parts-driven" approach. Development iterates between the two approaches. Information from previously developed, related applications is used to guide top-down development. Using related applications, or "domains" as the source of reusable entities makes reuse more natural and achievable because the same types of objects are handled and the same types of problems are solved. A particular domain could mature to the point that developers have templates for every stage of the life cycle and need only fill in the details that pertain to the particular application at hand.

"A domain-oriented life cycle formalizes typical patterns in the development of a related series of applications and the persistence of information from one application to the next." It requires new methods of analysis to determine what parts of a particular application are appropriate for reuse, and how that information should be kept. For applications with significant amounts of redundant code, Simos proposes the use of Application-Specific Languages (ASLs). An ASL is a non-procedural specification language that has been developed specifically for a class of applications. A translator could be used to translate a specification in an ASL into the code of a high level language. Applications that can be used in a variety of situations (e.g. a statistical package) could be developed with a library approach. Research areas include exploring ways to decompose problems to determine appropriate domain classifications and the most effective methods of reuse. The domain-oriented life cycle then allows developers to "capture and reuse domain specific knowledge across applications, thereby accelerating and rendering manageable the informal process of reuse that currently occurs."

As can be observed from this summary, Simos' approach is primarily concerned with development. He makes no mention of the V&V phases of the life-cycle.

Reusability Guidebook

The Reusability Guidebook¹ is a compilation of the major issues related to reusability being studied by the Ada community. It does not specifically address the issues of verification and validation although it does refer to terms such as software testing and proofs of correctness. None of these occurrences directly relate to the V&V of reusable components. The main points mentioned in the document were the following 1) because reusable

components would be heavily used, they should thus be tested more thoroughly, and 2) the fact that they are used more and in more varied environments would eventually result in more reliable products.

An important issue that is pointed out by the Guidebook is that there are three different types of individuals who are involved in the reuse effort. They are the parts manufacturer, the parts user, and the librarian. Parts manufacturers have to verify and validate their products with greater care since they are likely to be used in a large number of application and machine environments, and the success of reuse will largely depend upon the confidence that the users will have on the reusable parts. The parts user should only, (once the applicability of the part is understood), have to be concerned with integration (black box) testing of the different parts. The librarian should have a well understood way of accepting or rejecting submitted parts.

A point made in the Guidebook is the need for metrics of reusability. The metrics should cover concepts such as a measure of confidence that a new user of a previously developed part may reasonably apply to the suitability of a part for a potential new application. There should also be metrics of the independence of the part from the host and target computer. Furthermore, metrics measuring characteristics such as coupling, cohesion, reliability, modifiability, localization, protection against incorrect usage, and error handling are also important and could be used to determine the adequateness of a part.

Although there are some well known metrics that can be used to measure characteristics such as coupling and cohesiveness, some of the other characteristics mentioned do not have adequate or generally accepted measures.

The issue of liabilities and warranties is also explored by the Guidebook. The point is made that a developer might be reluctant to reuse parts if he or she will be liable for failures of subsequent reuse. The part developer, the Guidebook suggests, would be responsible for guaranteeing that the part meets its original specification. The user of reusable parts would be responsible for the developed product, within the limits of any warranties which may exist from the original developers of the incorporated parts. There are three categories of ownership of parts: Government ownership, mixed ownership, and private ownership. Under Government ownership, use of parts is optional and any user accepts full responsibility for the use of the part. Under mixed ownership liability is governed by the terms

of a license agreement, and the owner accepts responsibility for meeting the part specification. Under private ownership liability is governed by the terms of a license agreement.

Software parts are normally designed in the context of a particular method such as object-oriented design or functional decomposition. Parts designed based on a single paradigm have certain distinct advantages over the multiple paradigm approach. The Guidebook notes advantages of designing parts based on a single paradigm. Specific goals mentioned in the Guidebook are the following: 1) uniformity of interface comprehension, 2) simplification of intra-component comprehension, 3) simplification of part development and maintenance, and 4) simplification of part testing and optimization for automated testbed generation.

The Guidebook states that integration of parts in unanticipated ways requires a larger testing effort. Part of any reusability effort should go into developing a well supported mechanism to integrate reusable parts. Unfortunately, the Guidebook is unable to define a methodology to achieve such objectives.

Several other points are made in the Guidebook that are worth commenting on here.

- + "The use of proven parts can reduce levels of development effort and test and integration time through fewer errors and will result in more reliable products."

Even though this is a logical statement, it is important to realize that parts still need to be used for their original purpose and that if they are not, they may be the ones causing the "faults" in the system. Assuming that they are correct under any circumstance can only create problems.

- + "The use of proven parts can reduce levels of development effort."

This, of course, is dependent whether the part is being used in a way anticipated by the original developers and on how much adaptation needs to be performed.

- + "Testing is facilitated by use of one intensively tested generation piece." Once this piece is trusted, only variable parametric attributes of each instance need to be tested further.

It may not be possible to parameterize all of the desirable dimensions of reuse, at least where Ada generics are concerned.

Conclusions and Recommendations

Traditional software life cycle models do not address the topic of reuse. Those people who have considered the issues of reuse have largely concentrated on the implementation phase of software development. We believe that incorporating stages into life cycle models where the primary motivation is the design or incorporation of reusable components would be highly beneficial. We also believe that some of the labor intensive activities involved in incorporating reuse could be automated by a software development environment. Although some attention has lately been paid to the reuse of designs, we are not aware of any discussion of the relation of reuse to the V&V activities of the software development life cycle.

When reuse is a consideration, V&V activities becomes more complicated. The complications arise from differences between the environment for which a component was developed and the one in which it will be reused. Environmental differences derive from different hardware architectures, compilers, run-time systems and from different application environments and usage patterns. Traditional V&V treats these issues as non-functional requirements. Conventional testing techniques are constrained to detecting differences between functional requirements and actual program behavior and therefore can not be applied to non-functional requirements.

To deal with these problems, traditional functional specification techniques need to be extended to deal with these environmental issues. The first step to accomplishing this is to characterize the environmental constraints that may affect the behavior of a component. Such a characterization includes language issues and a discussion of application environment issues such as synchronization and memory management. A description of an initial characterization has been given in a separate paper.

References

- [1] Berard, E. V., "Creating Reusable Ada Software," *Proceedings of the National Conference on Software Reusability and Maintainability*, September 10-11, 1986.
- [2] Boehm, B. W., "A Spiral Model of Software Development and Enhancement," *Proceedings of the IEEE Second Software Process Workshop, ACM Software Engineering Notes*, August 1986.

[3] Bullard, C. K., Guindi, D. S., Ligon, W. B., McCracken, W. M. and Rugaber, S., "Verification and Validation of Reusable Ada Components," *Proceedings of the Sixth Empirical Foundations of Information and Software Sciences*, 1988.

[4] Department of Defense, "Defense System Software Development," MIL-STD-2167, Superintendent of Documents, U.S. Government Printing Office, Washington, D.C., 1985.

[5] Department of Defense, "Defense System Software Development," MIL-STD-2167a, Superintendent of Documents, U.S. Government Printing Office, Washington, D.C., 1988.

[6] Emerson, T. J., "A Discriminant Metric for Module Cohesion," *IEEE*, 1984.

[7] Guindi, D. S., Ligon, W. B., McCracken, W. M., Rugaber, S., "The Impact of Verification and Validation of Reusable Components on Software Productivity", to appear in the *Proceedings of the Hawaii International Conference on System Sciences*, January 1989, Kailua-Kona, Hawaii.

[8] Matsumoto, Y., "Management of Industrial Software Production", *IEEE Computer*, February, 1984.

[9] McNicholl, D. G., et al., "Common Ada Missile Packages," Technical Report AFATL-TR-85-17, Eglin Air Force Base, Florida, June 1985.

[10] Simos, M., "The Domain-Oriented Software Life Cycle: Towards an Extended Process Model for Reusability," *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse*, 1987.

[11] Wald, E., et al., "Reusability Guidebook V4.2," STARS Program, 1986.

[12] Wolverton, R. W., "The Cost of Developing Large-scale Software," *IEEE Transactions on Computers*, Volume 23, Number 6, June 1974.

Biographies



Dany S. Guindi
Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Dany Guindi is a research scientist at the Software Engineering Research Center and the School of Information and Computer Science of the Georgia Institute of Technology. During his tenure he has worked on a variety of projects, including the Mothra project, a system for verifying the accuracy of programs based on "mutation theory", as well as a STARS funded project for studying the Verification and Validation Issues of Reusable Software Components. Dr. Guindi is also interested in Human Factors and has been involved in the design and development of the X Window system and the X Toolkit. He is currently working on the Video X project, an extension to X Windows to allow the imaging of moving video pictures under X.



W. Michael McCracken
Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

W. Michael McCracken is a Senior Research Engineer in the School of Information and Computer Science and is responsible for the technical program of the Software Engineering Research Center at the Georgia Institute of Technology. Mr. McCracken is currently the manager of the Program Mutation for SDI Applications project sponsored by the U.S. Air Force Rome Air Development Center. He is also the Principal Investigator of a project that is developing verification and validation methodologies and techniques for testing reusable Ada components for the DoD initiative, Software Technology for Adaptable Reliable Systems (STARS). In addition, he is the Principal Investigator of a project that is studying the impact of Integrated Services Digital Networks for the U.S. Army's Information Systems Command. In addition to the above projects, Mr. McCracken is the Principal Investigator and Co-Director of a new National Science Foundation Industrial University Cooperative Research Center for Information Management.



Spencer Rugaber
Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Spencer Rugaber is a research scientist in the Software Engineering Research Center and the School of Information and Computer Science of the Georgia Institute of Technology. His research interests include software maintenance, the software design process, and the design of programming languages. He is currently working on the development of a hypertext software maintenance system.

A LOGICAL FRAMEWORK FOR VERSION AND CONFIGURATION MANAGEMENT OF ADA COMPONENTS

A. T. Jazaa and O. P. Brereton

Computer Science Department, University of Keele, Keele, STAFFS ST5 5BG, U.K.

A logic framework for version control and configuration management of Ada programs is proposed. The paper describes the motivations and benefits of our approach and shows how cross-referential and dependency information can be abstracted from Ada compilations to support version and configuration management. A prototype logic-based program library, implemented in the logic programming language Prolog, is also described.

Introduction

The paper describes a new program library structure for Ada based on a logical framework for *version and configuration management* of software components. Prototype tools, written in *Prolog*, for extracting information from a parsed program, for creating new versions of a program, and for storing the information in the proposed program library are described. Before looking more closely at the program library and associated tools we discuss the general Ada compilation philosophy and outline some of the limitations of existing approaches to configuration management.

Library Units and Separate Compilation

The programming language Ada allows programs to be put together from a number of source texts that have been compiled separately. The text compiled on a single occasion is known as a *compilation*. Each *compilation* is a collection of one or more *compilation units*. A *compilation unit* is a separately compiled specification or body of a subprogram or package, or a subunit.

A compilation unit is either a *library unit* or a *secondary unit*. A library unit is a compilation unit (which is a self-contained, independent module) that can act as a reusable software component or a building block for other software projects. A self-contained compilation unit is one that should not be dependent upon another module or data structure as far as possible. A library unit is a subprogram specification, a package specification, a generic specification, a subprogram body, or a generic instantiation. Each library unit must have a simple name that is a distinct identifier.

A compilation which is a subprogram body is interpreted as a secondary unit if the *program library* contains a unit with the

same name. It is otherwise interpreted both as a library unit and as the corresponding library unit body (that is, a secondary unit)¹. The effect of compiling a compilation unit that is a library unit is to define (or redefine) it as one that belongs to the *program library*.

A secondary unit is either the separately compiled proper body of a library unit, or a subunit of another compilation unit. The effect of compiling a compilation unit that is a secondary unit is to define the body of a library unit. The effect of compiling a secondary unit, as a subunit, is to define the proper body of a program unit that is declared within another compilation unit.

The secondary unit (implementation) of a library unit can be compiled and added to the *program library* at a later time. This means that the implementation of a library unit (such as the body of a subprogram or package, or subsequently a subunit) can be changed repeatedly without affecting any software that makes use of the library unit.

Subunits can be used to decompose a large-scale software project into manageable software components the implementation of which can be deferred to a later time. This type of software development supports the use of structured design techniques. By compiling a compilation unit that contains a number of stubs (that are specifications of program units), the compiler will be aware that some implementations will follow at a later time and may be stored in different files. A subunit construct has a number of advantages. These are:

- It provides support for top-down development
- entities of a subunit compilation can be taken down to any level required, enabling the developer to design the software structure more accurately.
- It can make use of library units, thus reducing the number of dependencies.

Information concerning a *compilation unit* is added to a *program library* when it successfully compiles. *Compilation units* stored in the *program library* can be used as components of several programs. A *program library* is a database for software development, therefore any separation of it from the software environment tools, can be considered to be impractical for large-scale projects.

The Main Program

All compilation units of a program must be stored in one or more *program libraries*. These compilation units are invoked by means of a *main* program unit that will link together all library units required to configure a software system. Ada requires that the root of every software system be a subprogram, since it

represents an algorithmic abstraction. *Main* programs are subprograms that are, at least, parameterless procedures. A *main* program must be a library unit. An implementation may impose certain requirements on the parameters and on the result, if any, of a *main* program.

In our case⁹, a *main* program unit is defined to satisfy the following conditions:

1. It is a subprogram body,
2. It is the root of a software system, and
3. It is parameterless^{4, 5}

Visibility of compilation units

Ada encourages the development of *self-contained* modules. These modules act as ready-made components for future re-use; hence an easy and flexible approach to incorporating these modules is required. Ada provides two constructs to make modules visible to other software components that need to use them. These constructs are:

- i- *with-clauses*, and
- ii- *use-clauses*.

With-clauses express relationships between compilation units. They specify which library units are being used or which other compilation units are necessary for the execution of a given unit within which the clause appears. They also enable the compiler to check the use of these library units (listed in the *with-clauses*) in the submitted compilation against their specification in the *program library*.

The second construct, *use-clause*, is used only with package units and enables package exports to be referenced directly by their names in the program unit under consideration. If *use-clause* is not specified the syntax:

< name-of-package > . < name-of-export >

would have to be used for every applied occurrence of an export. The use of *with-clauses* and *use-clauses* is an aid to writing programs that are readable and easily maintainable.

Software Configuration Management Background

Software products have been applied to many complex problems in the software industry throughout the 1970's.

Many of these products have failed over the last decade³. In response to these failures more attention has been given, within the software engineering community, to methods towards the automation of software configuration management.

Software Configuration Management (SCM), has been defined by IEEE standards¹⁰ as the process of:

- identifying and defining the configuration items in a system
- controlling the release and change of these items throughout the system life cycle
- recording and reporting the status of configuration items and change requests
- verifying the completeness and correctness of configuration items.

⁹Ada Compilers under UNIX.

The fundamental features of SCM are

- 1- the identification of components of a software product
- 2- the build dependencies inherent in each component
- 3- the build rules to derive or rederive executable code from source code.

Most existing software configuration and version management tools have the following limitations:

- only source code is covered by the storage and/or controlling schemes. No provisions are made for incorporating other objects such as specifications, documentation, or user requirements
- the possible relationships between versions of a module are limited in type and generally fixed in number
- it isn't possible to incorporate different styles of version management in different projects within an organization, and

For Ada, what tools exist have the following drawbacks:

- integration between a *program library* and the environment tools is difficult and inflexible
- the *program library* is complex, restrictive, and lacks portability
- each implementation has its own environment tools that can not be used by other implementations. The Ada programming language is portable and so the environment tools should also be portable
- dependencies and relationships between configuration components and versions are not computed automatically from the Ada source file. For example, in the DSEE system, the user has to provide dependency relationships among components⁸.

The proposed version and configuration management system aims to overcome the above drawbacks by providing:

- the power to incorporate intelligence in browsing and in command interpreting, along with a consistent representation of knowledge and data,
- a deductive capability of producing new facts from existing ones. A prototype implementation has been written in *Prolog*, which unlike relational databases, provides an inference mechanism^{11, 12},
- a portable *program library*, which depends only on the Ada syntax rules as defined by the Ada Reference Manual. It is implemented using an Ada parser written in *Prolog*, from which the necessary facts are extracted and stored in the *program library*,
- a complete integration between the *program library* and configuration and version management model,
- the possibility of having more than one *program library* such that there can be a high interaction amongst them,
- an automatic computation of *compilation* and *recompilation* dependencies from the Ada source code through the parser,
- a way to compile new versions without interpreting them as a recompilation unit,
- a capability, such that when a program unit has been modified an automatic request is generated, to modify all units affected by this modification. For example, if a specification of package is modified then an automatic

request to modify its dependencies (i.e its body and as a consequence the affected sub units of the modified body) is issued,

- a suitable method for expressing software component functionality for the purpose of re-use. Boehm⁷ has suggested that very significant improvements in software productivity will only be seen when software re-use is widely practised,
- a simple and efficient way of deleting old information, adding new information, and issuing queries about component names, types, storage locations, version numbers, time of first compilation, number of recompilations with dates, and so on, and
- a reduction in the size of the necessary code for building a programming support environment or version control and configuration management tools⁶.

The Program Library

A basic structure for CM for Ada is the *program library*. A prototype *program library* has been designed^{2,9} to act as a database for CM. The components of a *program library* represent structures within the software system being developed. These structures can be viewed as a set of components and a set of logical relationships and dependencies between the components. A logic programming language seems, therefore, an appropriate tool which can be used to express these logical relationships and dependencies.

We believe that this prototype *program library* offers a firm base for:

- building a more efficient and flexible *version and configuration management* system for the Ada programming language,
- expressing the functionality of the components of a library,
- portability, because it depends on the grammar rules of Ada as defined in the Ada programming Reference Manual.

The *program library* is created using the following tools:

- an Ada parser written in the logic programming language *Prolog*,
- the tool 'makelib' to insert the facts, extracted from the parsed Ada program, into the *program library*.

Dependencies and relationships between configuration components and versions are computed automatically from the Ada source file.

The current implementation has a tool 'create_version' to create new versions of a component by automatically copying or inheriting values of properties (i.e attributes) from a previous version onto the new created version. When a version of a component (or compilation unit) is created from scratch then the tool 'create_version' calls up the tool 'makelib' to parse the component source code so that the necessary information can be extracted and stored in the program library. This program library is similar to any text file under UNIX. The information extracted by the tool 'makelib' is represented by the following facts:

- 1- type of compilation unit such as:

```
package_specification( Compilation_Unit).
package_body( Compilation_Unit).
subprogram_specification( Compilation_Unit).
subprogram_body( Compilation_Unit).
generic_specification( Compilation_Unit).
generic_instantiation( Compilation_Unit).
```

- 2- dependencies such as:

```
parent_unit( Parent_Unit_Name, Sub_Unit_Name).
with_clause( List_Of_Units, Dependent_Unit).
```

- 3- other useful information such as:

```
composed_of( Compilation_Unit, List_Of_Units).
```

composed_of is a fact that shows the program units constituting a compilation unit.

The tool 'create_version' also provides a mechanism to add new information to the existing program library. This information includes the following sample of facts:

```
unit( optimise, optimise198831112345,
subprogram_body( optimise, optimise198831112345)).

file_name( subprogram_body31112345,
subprogram_body( optimise, optimise198831112345)).

date( optimise198831112345, [ 1988, 'Oct', 31, 11, 23,
45]).

system( optimise198831112345, ' UNIX').

language( optimise198831112345, ' Ada').

debug( optimise198831112345, reliable).

state( optimise198831112345, undetermined).
```

The Version Model

During the development and maintenance of a software product a number of versions of a particular module (specification) are generally produced. Different versions may be alternatives or variations that are applicable to different operating environments or systems or they may be revisions which are successive attempts to improve an implementation. Multiple versions may also exist for other reasons (for example a debug, a high speed/ high storage version) where the versions bear no chronological relation to each other.

We aim to encompass any relationship or set of version attributes that may be required, and do not impose any of the limitations which exist in other version management systems.

The value of a logical framework for version management is that it allows the user to reason about versions and to query the version database to, for example, identify a specific version of a module, or discover the relationships between all existing versions of a particular module. For example, one can find the version which is derived from one of the versions of the compilation unit 'optimise' with the time attribute constrained between Time1 and Time2.

```

derived_from( Compilation_Unit, New_Version, (Time1, Time2)):-
    unit( Compilation_Unit, Old_Version, _),
    copied_from( New_Version, Old_Version),
    date( New_Version, Time),
    Time < Time1,
    Time > Time2.

```

Deleting versions of an object can easily be achieved by searching for the required version either by giving its full structure or by including some of its attributes. For example:

```
delete( Version, Time ):-
```

```

    date( Version, Y),
    Y = Time,
    remove( Version).

```

This approach also provides a basis for managing the proliferation of versions during the course of a project. Automated clean up support can be provided. For example, one could delete all but a designated 'most recent' version, or all versions created before a specified time or within a period of time or all versions with a particular set of attribute values. In fact, any policy or convention that can be expressed as a rule could be enforced. It might also be desirable to implement a policy whereby attributes of an object could be modified without creating a new version.

The proposed framework covers not just source code but incorporates other objects in the software engineering life cycle such as user-requirements, specifications, and design. For simplicity, within our prototype implementation, user-requirements, specifications, and designs are made available through an object called *description*. As many revisions of the object *description* as required can be created. These revisions are linked directly to the source code (implementation) revisions through the fact:

```
description_spec( Description_Revision, Code_Revision).
```

It is possible to issue any query regarding description revision and code revisions. For example, one can issue a query such as:

```
list all code revisions of description version 'description198827124513'
```

The answer would be:

```

package_specification
('Iterate_Swan_Method','Iterate_Swan_Method198827034153')
package_specification
('Iterate_Swan_Method','Iterate_Swan_Method198828101932')
package_specification
('Iterate_Swan_Method','Iterate_Swan_Method198831122132')
package_body
('Iterate_Swan_Method','Iterate_Swan_Method198828107645')
package_body
('Iterate_Swan_Method','Iterate_Swan_Method198829324133')
package_body
('Iterate_Swan_Method','Iterate_Swan_Method198830112941')
package_body
('Iterate_Swan_Method','Iterate_Swan_Method198831111412')

```

From the database set of facts and the logical relationships that exist between compilation units and/or program units of the

Ada programming language, one can deduce new facts from existing ones, delete old information, add new information, and issue queries about component names, types, storage locations, version numbers, time of first compilation or recompilation with dates. For example, one can define a *main* unit using the following rule:

```

main( Unit ) :- subprogram_body( Unit),
                 root_of_system( Unit),
                 parameterless( Unit).

```

or issue the following query:

what are the versions of the component 'Complex_Relations' that were created on 31 October 1988 at or after half past nine in the morning ?

```

[ ?- unit( 'Complex_Relations', Version, Type),
    date( Version, Date),
    Date = [ 1988, 'Oct', 31, H, M, _],
    H > 9, M >= 30.

```

```

Version = 'Complex_Relations19883194144',
Type = package_specification( 'Complex_Relations',
'Complex_Relations19883194144').

```

one could ask "where is the above version stored?"

```
[ ?- file_name( File, 'Complex_Relations19883194144').
```

```
File = spec3194144.11
```

The current prototype system has a capability, such that when a program unit has been modified a request to modify all units affected by this modification is generated. For example, if a specification of a unit is modified then an automatic request to modify its dependences is issued. Two ways of notifying the user are:

1. relying on the user to issue a query:

"display the program units that constitute the modified compilation unit 'Complex_Relations'"

```
[ ?- composed_of( body( 'Complex_Relations', _), List).
```

The answer would be:

```

List = [ subprogram_stub( sqr, sqr198816121309),
        subprogram_stub( divide, divide198818321245),
        subprogram_stub( scalmult, scalmult19881089234),
        ...
    ]

```

2. automatically:

for example, when a new version of the package specification 'Complex_Relations' is created using the tool 'create_version' as follows:

| ? create_version('Complex_Relations').

the user is automatically notified of the dependencies by a response such as:

"Please remember to modify the corresponding body which is stored in the file 'comp_rel.A'".

When the file 'comp_rel.A' is modified a request is also generated to modify its stubs:

"Please remember to modify the stubs of the modified body which are stored in files 'rest_comp.A', 'sqr.A'".
then the system asks:

"Would you like to edit these files ? (y/n): "

Other version or object meta-information that could be recorded includes the reasons for its creation, the reasons for its deletion, ownership, derivation, and other statistical data. This information can be stored in the database as facts in the form:

```
creation( Object, Why),
deletion( Object, Reasons),
derivation( Object, Information),
modification( Object, Modify),
other( Object, Other).
```

This information could be useful, for example, for tracing the history of the implementation of a particular program specification. In fact systems that maintain this sort of information have proved useful in the past². Babich¹³ reports that

"many times the fastest approach to finding a bug is not analysis of the program itself, but analysis of the history of the program - how it was created".

Meta-data may record the textual derivation of an object or version. This could provide valuable information for storage algorithms which attempt to minimise the storage requirements of multiple versions of program.

Other possible queries about versions are:

```
list all versions of library unit X,
what are the components of a program?
find the stubs of a certain body unit,
find all versions of secondary unit Y that are created
between time1 and time2,
and so on.
```

This information will permit the building of portable version control and configuration management tools since the program library itself is portable.

Configuration of Software Systems

The Ada programming language is designed to support separate compilation for constructing large programs and creating program libraries of precompiled components. Compilation units can be compiled in any order as long as the checking of consistency between the compilation units and code generation is observed.

Our prototype configuration manager automatically compiles an Ada system (program) according to the rules governing the dependencies of compilation. When a compilation unit is submitted to the compiler, all its dependencies are identified and

compiled according to Ada rules⁹. A preset condition (or a set of conditions) is used to govern the selection of the required version or set of versions of each compilation unit.

The selection criteria can be variable, fixed, or under user or manager control, according to local needs. The following code is an example set of conditions for selecting a particular version of a compilation unit.

```
k_valid( Compilation_Unit, From_Time1, To_Time2):-
date( Compilation_Unit, Time),
Time > From_Time1,
Time < To_Time2,
state( Compilation_Unit, good),
language( Compilation_Unit, 'Ada'),
debug( Compilation_Unit, reliable).
```

A 'Compilation_Unit' (or set of units) is selected if it is created between time 'From_Time1' and 'To_Time2', its state is good, it is written in Ada (versions of modules written in languages other than Ada could be incorporated), and it is believed to be reliable.

If no version satisfies the preset condition then either a request can be issued for permission to use a default version, or "missing compilation unit" is simply reported. The current implementation allows defaulting to either 'preferred' version or to the 'latest' version but any logical rule could be imposed. It also incorporates version priorities.

When a compilation unit is modified and then recompiled, the whole Ada program, of which it is a component, does not have to be recompiled. The units that need to be recompiled are identified according to the following rules:

- A library unit requires the recompilation of all the compilation units that depend upon (i.e use) this library unit.
- If a compilation unit is not a library unit and consists of a package body or subprogram body, it only requires the recompilation of the subunits declared within its body. Other compilation units which use the modified compilation unit do not need to be recompiled, because they do not depend upon the implementation of the package or subprogram body.
- A subunit does not require the recompilation of a parent-unit or any other subunits.

When a unit is submitted to a compiler, the CM system will search the database for it. If a unit with the same name has already been compiled then the system will take a recompilation action, otherwise it will take compiling action. For the recompilation action, the system will automatically recompile all the necessary units according to the Ada recompilation rules. When a compilation unit successfully compiles the proposed configuration manager stores its type as a fact in the form:

```
compiled( subprogram_body( optimise, optimise198810111234)).
```

This additional information about a compiled unit allows new versions of a unit to be represented without them being treated as recompilation units (and hence replacing the version in

used). A preset condition can also be used to govern the selection of the appropriate version for recompilation.

Conclusions

Software configuration management and version control is an important aspect of software engineering. It provides the means of identifying the components, and the relationships between components, of a system at any point in time. This allows the systematic control of changes to a configuration and provides overall control, visibility, and traceability of a configuration throughout the life cycle of a software system.

A logical structure for the Ada program library and the configuration of versions has been proposed. This structure is created using an Ada programming language parser written in the logic programming language Prolog. The input to the parser is the Ada source file. Useful facts are extracted from the parsed program and written in the proposed program library using tools written in Prolog. This proposed system, we believe, provides the flexibility required to tackle the issues described above.

References

- [1]. "Department of Defense, Reference Manual for the Ada Programming Language," ANSI/MIL-STD 1815 January 1983.
- [2]. T. Lyons and J. Nissen (ED.), "Selecting an Ada Environment," Cambridge University Press, 1986.
- [3]. E.H. Bertoff, "Elements of Software Configuration Management", IEEE Transaction on Software Engineering, Vol. SE-10, No.1, January 1984.
- [4]. J. S. Briggs, "Two Implementations of the Ada Program Library", Software-Practice and Experience, Vol. 14(5), 491-500, May 1984.
- [5]. N. Gehani, "UNIX Ada Programming", AT&Bell Laboratories, Prentice-Hall, INC., Englewood Cliffs, NJ07632, 1987.
- [6]. P. Asirelli and P. Inverardi, "A logic database to support configuration management in Ada, Ada-Components: libraries and tools", proceedings of the Ada_Europe International Conference, Stockholm, 26-28 May 1987.
- [7]. B. W. Boehm, "Software Engineering Economics", IEEE Transaction on Software Engineering, Vol. SE-10, No. 1, January 1984, PP 5-21.
- [8]. David B. Leblang and Robert P. Chase, Jr, "Computer-Aided Software Engineering in Distributed Workstation Environment", ACM Sigplan Notices, 19, 5, May 1984
- [9]. A. T. Jazaa and O. P. Brereton, "A Configuration Management Framework for Ada", Proceedings of the seventh Ada International Conference, York, UK 19-21 September 1988.
- [10]. "IEEE Standard for Software Configuration Management Plans", IEEE Std. 828-1983, IEEE Comput.Soc., New York, NY, 1983.
- [11]. H. Gallaire and J. Minker, (1984), "Logic and Databases: a deductive approach", Computing Surveys, 16, (2), pp. 153-185, 1984.
- [12]. P. Asirelli, et al. (1986), "The Knowledge Base Approach in the Epsilon Project", In ESPRIT'85: Status Report of Continuing Work, The Commission of European Communities (Eds.), Elsevier Science Pub.B. V., (North Holland), 1986.
- [13]. W. A. Nablach, "Software Configuration Management: Coordination for Team Productivity", Addison-Wesley Publishing Company, 1986.

Dr. Pearl Brereton holds an established IT Senior Research Fellowship in the Department of Computer Science at Keele University. She joined the department in 1979 and has since worked on a number of Government funded projects. She currently heads the Systems Research Group and holds a major Alvey award investigating the use of

knowledge-based techniques in software engineering. Previously she has worked at The Science and Engineering Research Council's Daresbury Laboratory, at Nottingham University and for Pilkington Brothers Ltd.



Abid Jazaa is a postgraduate student involved with developing a logical configuration management system for Ada. He obtained his BSc from the University of London (Queen Mary College) in 1977 and MSc from Southampton University in 1978. He then joined the Iraqi Northern Petroleum Company as a researcher.

He is a member of the British Computer Society and a member of UK Ada.



DESIGNING FOR CHANGE : AN ADA DESIGN TUTORIAL

James A. Hager

HRB - Systems Inc.

Abstract

Sixty percent of the software costs associated with the design, development and implementation of computer systems occurs in the maintenance phase. A significant reduction in the maintenance costs can be realized with a design for change philosophy integrated into the Engineering Life-Cycle. By carefully identifying the expected changes to a system and rigorously applying the concepts of information hiding and abstraction of interfaces, the changeable aspects of a system can be isolated. This paper provides an Ada based design tutorial, by tracing the design process and the resulting architecture based upon these concepts.

Introduction

The widespread use of computers over the last 25 years has had pronounced effects within the Department of Defense. It is currently estimated that the DoD spends about 3 to 4 percent of its budget, or approximately \$10 billion dollars per year, on software. This number is expected to increase rapidly in the next few years¹.

Unfortunately, current methodologies for specifying, designing, documenting, coding, and testing software do not provide adequate visibility to maintenance concerns.^{2,3,4,5} The difficulty of generating software that is easily modified becomes evident when the full Engineering Life-Cycle costs are examined. Figure 1 graphically portrays the distribution of effort in the software Life-Cycle.⁶

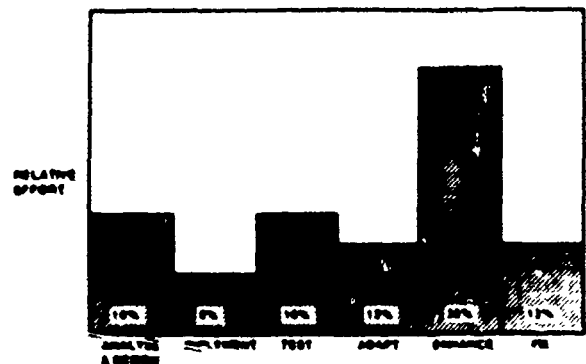


Figure 1. Distribution of Effort in the Software Life-Cycle

Several facts are apparent. First, software maintenance costs more than software development activities. Software maintenance involves three types of activity : enhancing the capability of a product, adapting the product to new processing environments, and correcting bugs.

Second, a large percentage of the total software effort is devoted to software enhancements.

In recent years, several new design methodologies and supporting languages have emerged whose goals are to reduce the overall costs associated with system development by reducing maintenance costs and providing more visibility to maintenance concerns during system life-cycle activities.^{7,8,9,10,11,12}

Software Cost Reduction Programs

In 1970, a Software Cost Reduction (SCR) program was initiated by the Naval Research Laboratory that pursued these software engineering project goals.

The SCR methodology requires changes in both the design methodology and the supporting documentation structures. Key SCR concepts upon which specification and design techniques are based include:

- 1) Separation of Concerns
- 2) Formal Specification
- 3) Information Hiding / Abstraction of Interfaces
- 4) Documentation as a Software Design Medium.

This paper focuses on the role of information hiding / abstraction of interfaces in the reduction of life cycle costs.

Information Hiding/Abstraction of Interfaces

Information hiding is a concept developed by Parnas in 1971.³ When a system is designed using information hiding as a decomposition criterion, design begins with a series of difficult decisions. Difficult design decisions are characterized by impacts that affect more than one module. Each module is designed to hide such a decision from the other modules. Each module in the system hides the internal details of its processing activities, and modules communicate through well-defined interfaces. Unlike functional decomposition, where changeable aspects of the system may span several modules, decomposition is structured so that high-probability changes do not affect the interfaces of widely-used modules. Less probable changes may affect the interfaces of small closely-held modules. Only very unlikely changes may affect the interfaces of widely-used modules.

Abstraction is a tool that allows one to deal with concepts apart from the particular instances of those concepts. All representation and manipulation details are suppressed. Objects of an abstract type are known only by the functions that may be performed on them. Users of the abstraction do not have access to the internal details of the abstract types.

The SCR design methodology is a process in which:

- 1) All expected changes are identified and prioritized early in the design process.
- 2) Information hiding and abstraction of interfaces are applied rigorously during the decomposition of the system into software modules.

Identifying the expected changes is a difficult process that requires significant familiarity with the application. Once the expected changes are agreed upon, they are prioritized based on their likelihood of occurrence. Although all expected changes are important, outside factors may prohibit application of the entire list. Prioritization of the expected changes allows some flexibility in this decision process. These changeable aspects of the system become the secrets of separate modules, thus providing a layer of insulation between the changes and the remaining software.

TRAINING SYSTEM BACKGROUND

In 1984, NRB-System's Inc. was awarded a contract to provide computer-based training for a large signal collection and processing system. The target system had a history of frequent and significant upgrades. The initial training system contract was awarded based on the success of a prototype that demonstrated the feasibility of enhancing training by means of computer-aided instruction.

Following a successful System Definition phase, the government redirected the effort to use the SCR methodology. This redirection was based upon the reduced risk associated with the existence of a working prototype and the desire to apply the methodology in the generation of a new system. To support initial efforts, SCR research materials were provided. Although these documents were not complete, they provided an adequate starting point from which to explore the methodology. Subsequently, the methodology was enhanced and evolved as the Software Engineering Principles (SEP) process.

Expected Changes

Expected changes are changes which are, or appear to be logical evolutions of the system. Based on customer inputs, expected changes are identified and prioritized during the System Definition phase and presented for review during the System Requirements Review. Following customer approval, the expected changes are included in standard requirements specifications and designers are held accountable for an architecture that supports these concerns. Architecture documents provide a mapping between the expected changes and the modules impacted by their implementation.

The following list is a subset of the expected changes identified during the System Definition phase for the computer based training system :

- terminal interface
- underlying operating system
- networking environment
- target system messages and displays
- student evaluation criteria
- student monitoring formats
- authoring exchange necessary to create/modify scenarios
- number and characteristics of the systems being simulated
- specifications for key data structures
- access policies for key data structures
- run-time environment
- language implementation
- additional authors, students and instructors
- additional classroom management tools

The expected change list was generated by reviewing modifications made to the target systems during the previous 5 years and by extensive interviews with customer representatives. To provide rapid access by maintenance personnel to areas of concern within the documentation, the expected changes were grouped in the following way :

- hardware related
- requirement related
- implementation related

These initial groups provided a starting point for architecture efforts.

Computer Based Training Architecture

The modularization strategy described above leads to a hierarchical structure in which each higher-level module hides the design decisions encompassed by its descendents. Based on the expected change list, a four level architecture was derived. The first two levels are generic in nature and would probably apply to any system. Levels one and two are logical groupings while levels three and four are physical and correspond to Ada packages.

The first level of the hierarchy consists of three modules:

- the Hardware-Hiding module
- the Behavior-Hiding module
- the Software Decision-Hiding module

Hardware-Hiding Module

The Hardware-Hiding module consist of modules that need to be modified if any of the hardware is replaced with a new unit with a different hardware/software interface but with the same general capabilities.

The Hardware-Hiding module is further decomposed into the Extended Computer and the Device Interface modules. The Extended Computer module hides those characteristics of the hardware/software interface that are likely to change if the computer is modified or replaced. It implements virtual hardware that is used by the remaining software. In particular, this module supports operating system related expected changes by hiding the underlying operating system. By specifying primitives for an operating system in the Virtual Operating System module, the remaining modules are insulated from changes to the operating system. Primitives are fundamental assumptions located in the package specification that are very unlikely to change.

The Device Interface module satisfies the network and terminal related expected changes through the Virtual Network Interface and the Virtual Terminal modules. The Virtual Network Interface module hides the commercial network software and how the functions are made available to the system. Changes to the network hardware and software are insulated from the system application packages by identifying network primitives. Networking primitives include such services as establishing a circuit, sending a message, and receiving a message. If the methodology for establishing a circuit or sending a message changes, modules using the service do not have to be modified.

The Virtual Terminal module insulates the system from changes to the terminal by providing primitives for screen display and keyboard drivers. The display output device is managed as a set of windows, each with characteristics to simulate portions of target screen displays. The virtual interface provides the capability to radically change screen characteristics without affecting existing software. The Virtual Terminal interface hides the physical characteristics of the display device, locations of the devices, and windowing mechanisms.

Figure 2. provides a block diagram of the Hardware-Hiding Modules.

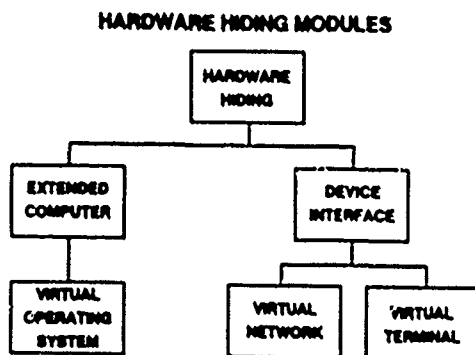


Figure 2. Hardware-Hiding Modules

Behavior-Hiding Modules

The Behavior-Hiding modules are the modules that need to be modified if there are changes to the required system behavior. Required system behavior is documented in the System Requirements Specification.

The Behavior-Hiding module is decomposed into two second level modules : the Application Driver module and the Shared Service module.

The Application Driver modules are the sole controllers of a set of closely related outputs. Each module hides the rules determining the values of the outputs and the data structures and algorithms necessary to implement the outputs. Expected changes dealt with in the Application Driver modules include the authoring exchange necessary to create and maintain scenarios, the system administrator exchange necessary to maintain target system databases, the system administration classroom management policies, the processing unique to specific target system simulations, the student evaluation processing and criteria, and the student monitoring processing. Impacts to the architecture based on these changes are restricted to a single module.

The Shared Service modules consist of software that controls required external behavior common to two or more modules. These modules hide the characteristics of the shared behavior and the algorithms and data structures necessary to implement the shared behavior. The Shared Service modules support expected changes related to module initialization, menu services, and control structures common to the modules. A change in any of these areas is isolated to the Shared Service modules, even though the change may affect an external behavior shared by many application modules.

It should be noted that all required system behavior is provided by the Behavior-Hiding modules. During preliminary architecture efforts, design credibility is established by mapping the required system behavior identified in the System Requirements Specification to these modules. Any requirements not mapped to a module or mistakenly mapped to a Hardware-Hiding or Software-Decision Hiding module provide areas to revisit the system architecture. Several discrepancies were noted in this manner.

Figure 3. provides a block diagram of the Behavior-Hiding modules.

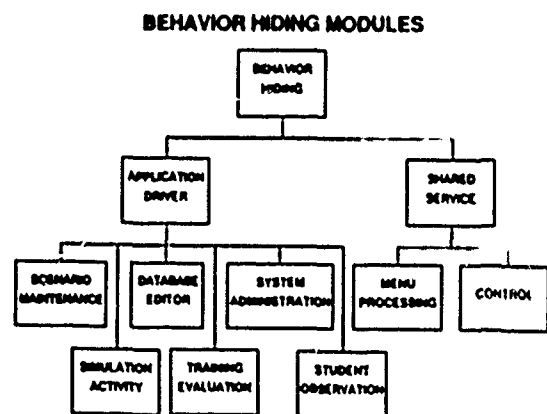


Figure 3. Behavior-Hiding Modules

Software Decision-Hiding

Software Decision-Hiding modules are the modules that need to be modified if there are changes to designer generated decisions. For example, the choice of a specific algorithm not specified in the System Requirements Specification is a designer generated decision.

The Software Decision-Hiding module is decomposed into three second-level modules: the Scenario Interface module, the Database Utilities module, and the System Generation module.

The Scenario Interface module hides changes to the scenario validation policies, the translation process from the external scenario language utilized by the authors to the internal scenario primitives, and the execution of those primitives. All algorithms to parse, validate, translate, and execute the scenarios are hidden in these modules. These changes were allocated to Software Decision-Hiding modules because the specific language implementation necessary to support required system behavior was designer determined.

The Database Utilities module consists of software that needs to be modified if changes are made to the database management system or to the internal storage, retrieval or maintenance policies. To insulate application modules from the underlying database management system, a Virtual Database Interface module is provided. It provides the file management primitives necessary to support indexed sequential access data retrieval. Any changes to the data access policies are limited to this module.

The System Generation modules hides the expected changes related to the software processing environment and the underlying language. It hides the command structures necessary to compile and link the software, values of system generation parameters that select different implementations of a module, and specialized test software.

The Language Implementation module provides an area to discuss features unique to the specific implementation chosen. Originally, the goal was to abstract out the underlying language implementation. Since this was cost prohibitive, it provided an area to discuss the language specific decisions that might affect program portability.

Figure 4. provides a block diagram of the Software Decision-Hiding modules.

SOFTWARE DECISION HIDING MODULES

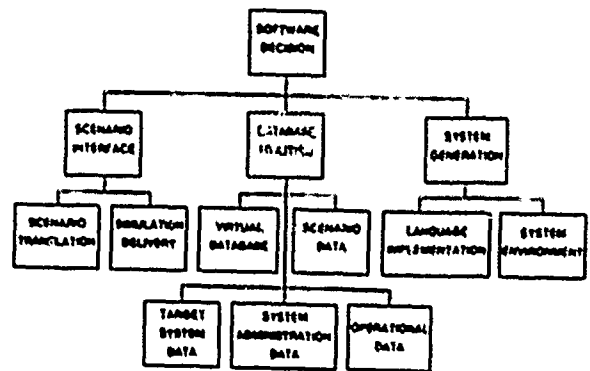


Figure 4. Software Decision-Hiding Modules

Translation To Specification

Designing a software system involves three tasks.¹⁵ The first is decomposing the system into modules to support system requirements. This was discussed in the first part of this paper. The second is designing the interface of each module. The third is producing a specification for each interface so that (a) implementers have enough information to write the software; (b) writers of other modules have enough information to use the module; and (c) information that constrains or discloses details of the implementation is not revealed.

Following the decomposition theme employed during the Architecture phases, it is important to design the interface of each module so that it consists only of information about a module that is not likely to change. In that way, when changes that affect a module are required, only the implementation of that module is likely to require a change. The interface and all other modules that use the interface are not likely to change.

Although the mapping between the module decomposition and the Ada packages necessary to support the module decomposition is straightforward, it is not always clear how to translate required system functionality into package access functions. The most effective approach is to focus on the Hardware-Hiding modules. Since the inputs to and the outputs from these devices are discrete or well known, access functions are directly associated with outputs. Abstract interfaces for the Virtual Operating System, Virtual Terminal, and the Virtual Network modules were generated in this manner.

For example, the Virtual Terminal package specification contains access functions to support basic terminal functionality, i.e., scroll, blink, highlight, color, cursor positioning, etc. Each functional abstraction should not reveal characteristics dependent on the underlying terminal implementation. It is expected that some of the functionality specified in the interface would not be used by the specific application, but necessary to fully characterize an abstract terminal interface. These "unused" functions would have empty implementations.

Some Software Decision-Hiding module interfaces were determined in a similar manner. For example, the Virtual Database interface was determined by consulting commercial database technical references and allocating an access function for each service required (get record, insert record, delete record, etc.). The integrity of the underlying data abstractions were enforced by relying on Ada's private data structure support.

The Behavior-Hiding module interfaces were generated by looking at which system level functions were allocated to each module. In some cases, there was a one-to-one mapping between system level functions and module access functions. In other cases, several system level functions were combined into one access function with the input parameters controlling the required output.

Figures 5, 6 and 7 provide graphical representation of the Virtual Terminal, Virtual Database, and the System Administration Data specification packages.

The SEP methodology, like any design process, does not provide a "cookbook" methodology to transition from specification to design. By focusing on the Hardware-Hiding modules, with discrete inputs and outputs, and the modules designers are most familiar with, a good portion of the interface functionality can be specified.

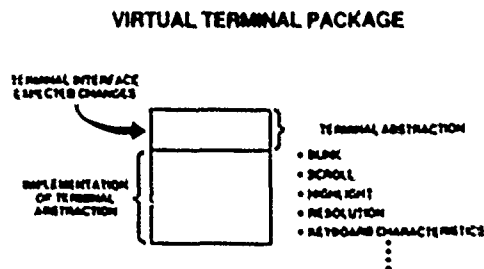


Figure 5. Virtual Terminal Package

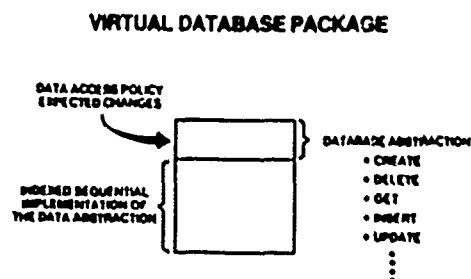


Figure 6. Virtual Database Package

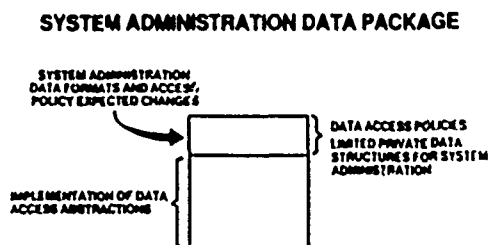


Figure 7. System Administration Data Package

The specification of an abstract interface should have the following properties¹⁵ :

- it must not disclose any of the changeable aspects of module
- it must present a concise description of the facilities available from a module in terms of effects that are directly observable to the user
- it should be divided into sections and formatted so that a reader unfamiliar with the module is able to find a piece of information without having to study the entire interface specification
- it should not provide duplication of information which would make using and maintaining the document more difficult

A complete example of a module specification satisfying these requirements and the associated guidelines for generation is found in reference (14).

Summary

The ultimate goal of the SEP design methodology is the reduction of costs associated with the production and maintenance of software systems. By providing more visibility to maintenance concerns at each phase of the product development, engineers are better able to plan for the expected system changes. It is too early to judge the success of the methodology at this level. Several years of accurate life-cycle cost data are required to support this premise.

However, there have been some immediate benefits to the methodology. From a design perspective, the methodology has proved very successful in reducing near term Engineering Life-Cycle costs. During a recent upgrade to the system, three significant target system simulations were added. Since these simulations were identified early as expected changes, designers were able to minimize the impacts of these changes on the existing architecture and documentation structures. Each simulation was added as a fourth level module under the Simulation Activity module. Commensurately, the original system delivery dates have been accelerated and cost reduced to reflect this reduction in effort.

A second update consisted of replacing the existing 8088 based workstations with 80286 based workstations. Although the new workstations provided higher resolution monitors and enhanced keyboards, the impacts were confined to the Virtual Terminal module.

A final update consisted of replacing the existing removable storage media with fixed internal storage. This was necessary due to the unreliability of the disk drive. The decision to replace the faulty drives was made following the successful completion of the Software Integration and Testing phase. Although the impact to the operations concept was significant, the software modifications were confined to the Control module and the Data Specification modules.

In all cases, the impacts would have been more severe without a design philosophy that required engineers to plan for these changes.

REFERENCES

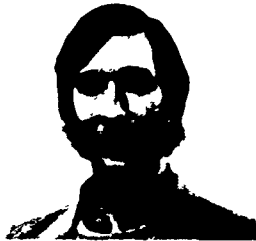
1. R.L. Fairley, Software Engineering Concepts, McGraw-Hill Co., 1985
2. P. Clements, R. Parker, D. Parnas, and J. Shore, A Standard Organization for Specifying Abstract Interfaces, NRL Report 8815, 14 June 1984.
3. D. Parnas, P. Clements, and D. Weiss, "Enhancing Reusability with Information Hiding", Proceedings of the Workshop on Reusability in Programming, pp.240-247, 7-9 September 1983.
4. K. Britton, and D. Parnas, A7-E Software Module Guide, NRL Memorandum report 4702, 8 December 1981.
5. K. Heninger, Specifying Software Requirements for Complex Systems: New Techniques and Their Application, NRL Memorandum, April 1979.
6. B. Boehm, "Software and Its Impact: A Quantitative Assessment", Datamation, May 1973.
7. J.A. Hager, "Designing For Change", Proceedings of the 7th NSIA International Conference, May 1987.
8. K. Heninger, J. Kallander, D. Parnas, and J. Shore, Software Requirements for the A7-E Aircraft, NRL Memorandum Report 3876, November 1978.
9. R.J.A. Buhr, System Design with Ada, Prentice-Hall, Inc., 1984
10. B. Liskov, and J.V. Guttag, Abstraction and Specification Program Development, MIT Press, Cambridge, Mass., and McGraw-Hill, New York, March 1986.
11. Berzins, Gray, and Naumann, Abstraction-Based Software Development, Communications of the ACM, May 1986.

12. P. Clements, Software Cost Reduction Through Disciplined Design, NRL Memorandum, 22 Feb. 1985.

13. Wallace, Stockenberg, Charette, A Unified Methodology for Developing Systems, McGraw-Hill. 1987.

14. J. A. Hager, Software Engineering Principles Study Report, NRL Report, April, 1988

15. P. Clements, R. Parker, D. Parnas, J. Shore, A Standard Organization For Specifying Abstract Interfaces, NRL Report, June 14, 1984.



James A. Hager
1627 Oxford Circle
State College, Pa. 16803

James A. Hager is a Principal Engineer for HRB-Systems in State College, Pennsylvania. He has had fifteen years experience, holding both technical and management positions, with the responsibility for the development of software for engineering products and systems. He is the author of numerous research articles relating to the design/development and maintenance of systems. He holds masters degrees in Computer Science and Mathematics from the Pennsylvania State University and is a member of the ACM, MAA, and IEEE.

A Portable Ada Implementation of Blocked_IO

John J. Cupak Jr., CCP
HRB Systems, Inc.

Keywords File management, blocked input-output, information hiding, data abstraction, packages.

Abstract

Efficient access to file elements is important when processing large quantities of topographical or terrain data. While the standard Direct_IO and Sequential_IO Ada packages permit access to individual file elements, they do not allow the user to read large blocks of data at a time and process individual elements.

This paper discusses a portable generic Ada package which gives the user access to single file elements, while efficiently reading/writing large blocks of data to and from memory. The concept of data abstraction is used to encapsulate the format of a blocked file, while information hiding is used to encapsulate the algorithmic implementation of the support procedures and functions.

Implementation and portability problems encountered are presented at the conclusion.

1 Ada File User Requirements

Terrain and topographical map data files can contain up to a million and a half bytes for a one degree square area. Using the standard Sequential_IO or Direct_IO package procedures to read three-thousand (3,000) 512-byte records, one at a time, would impose a file transfer time penalty on the processing of the data. For every file transfer request, the system incurs disk seek and latency time before transferring only a single record. One way to reduce the disk access time is to transfer more data at a time, thereby reducing the number of disk accesses. This could be accomplished by the programmer by declaring an array of data elements, instantiating an input-output package with the data array, and controlling reads and writes to the external file when the data array is empty or full. This is contrary to the concepts of data abstraction and information hiding, where the user should only be concerned with opening, reading, and writing individual records, and closing the external file. The actual implementation should be hidden from the user.

The map data files are usually written by one computer system and read on another. It was necessary, then, to define an implementation which could be used on at least

three different computers with three different operating systems, using three different Ada compilers. That is, the implementation had to be portable.

Map data files have many formats and contain various number of records. Writing a package to handle the blocking and deblocking operations for each data format would not only be time-consuming, but a duplication of effort. It would be better to write the operations once, encapsulate them in a package, and parameterize the package with the data record format and the number of records per data block.

The above description of the problem leads us to define the user requirements as:

- Reduction of disk access time
- Encapsulation of operations in a package
- Hidden blocked data structure
- Portable package
- Generic package

The following sections discuss the design of the generic parameters, the package user interface, and the implementation of the package.

2 User Interface

2.1 Design Consideration

The input-output facilities of Ada [LRM 83] are capable of handling sequential files by means of the Sequential_IO package, or random-access files by means of the Direct_IO package. It is not possible, however to specify that files be transferred as blocks of records. The only way to provide this capability is by means of a user-developed package. For sake of consistency, the package should conform to the structure of the existing input-output packages.

For example, the Sequential_IO and Direct_IO packages are implemented in similar manner. When only sequential access is required, the procedures in Direct_IO can be used in exactly the same way as those procedures in Sequential_IO. This concept of similarity was used in the implementation of the new Blocked_IO package. This similarity

supports the Software Engineering goal of Understandability. The more alike the procedures in the Blocked IO package are to those in the Direct IO package, the more the user will be able to understand their functionality and use them.

The blocked file organization allows the user to read and write individual logical records without regard to their position within the physical block they are in. The blocked file structure is thus determined by only two kinds of information: the kind of data element, and the blocking factor.

Translated into Ada notation, only two generic actual parameters must be supplied for any particular instantiation of the Blocked IO package: the **ELEMENT_TYPE** and the **BLOCKING_FACTOR**. The user has complete freedom to specify any data record, including variant records. The generic formal part of the Blocked IO package is written as shown in below

generic

```
type ELEMENT_TYPE is private;
-- Logical Record Type

BLOCKING_FACTOR : is POSITIVE;
-- Number of Logical Records per Block
```

2.2 Package Specification

Since similarity with the Direct IO package was desired, the specification part of the Blocked IO package is built in the same way as the specification part of Direct IO. The Blocked IO package exports:

- types
- exceptions
- file management operations
- input-output operations

There are a few basic differences between the packages. The Blocked IO package disallows the use of the **MODE** parameter since all operations are based on direct access.

The following sections discuss those items which differ from the specifications of Direct IO. A complete listing of the specification part for Blocked IO is given at the conclusion of this article.

Types

The **FILE_TYPE** specified for the Blocked IO package is the same, but the **FILE_MODE** is omitted, since all operations are based on direct access. The mode is always set to **INPUT_FILE**. The integer types **COUNT** and **POSITIVE_COUNT** used for file indices, are specified as in Direct IO.

Exceptions

All the exceptions of Direct IO are available except for **MODE_ERROR**, which is omitted since the user is not permitted to specify the mode in either a **CREATE** or **OPEN** operation.

File Management Operations

The **CREATE** and **OPEN** file management operations do not require the **FORM** parameter. Likewise, the **MODE** function is omitted. The remaining file management operations - **CLOSE**, **DELETE**, **RESET**, **NAME**, **FORM**, **SIZE** and **IS_OPEN** - are exactly the same as those defined for Direct IO.

There are three kinds of records manipulated by the package:

- logical records
- external records
- internal records

The user reads and writes a logical record using the Blocked IO operations. This is also the record type that is used to instantiate the Blocked IO package. Records of this type are numbered from one to the last record written by the user.

An **external record** is transferred to and from the external file. It consists of **BLOCKING_FACTOR** number of logical records. These records are numbered from one to the last "block" written by the Blocked IO Write procedure. The number of the last **external record** is returned by the **Size** function.

An **internal record** is one of the logical records contained in the internal **BUFFER**, which has been read in from an **external record**. The **internal record** number ranges from one to **BLOCKING_FACTOR**.

Three additional file management functions have been defined for the Blocked IO package to help the user identify which logical, internal, or external record is being manipulated. These functions replace the **Index** function available in the Direct IO package. These functions are named for the record types manipulated:

- Logical Records
- External Records
- Internal Records

These functions return the number of the external file record currently in memory, the number of the logical record in the file, and the number of the logical record in the external record, respectively.

Input-Output Operations

The input-output operations of the Blocked IO package are identical to those of the Direct IO package. There is no

Set Index procedure, however, as this capability is furnished by the Read and Write procedures when using the FROM and TO parameters.

2.3 Example of Usage

To illustrate the use of the Blocked_IO package, we will write a file of 256-byte records, blocked by four, resulting in an external file record whose size is 1024-bytes. The records will be written sequentially.

```
with BLOCKED_IO;
procedure TEST_DRIVER is

    type DATA_RECORD is array (1..64) of FLOAT;
    -- 64 elements x 4-bytes/element = 256 bytes

    package TEST_IO is
        new BLOCKED_IO
        (ELEMENT_TYPE    => DATA_RECORD,
         BLOCKING_FACTOR =>          4);

    -- This is the instantiation of the BLOCKED_IO
    -- package with the 256-byte DATA_RECORD, and
    -- a BLOCKING_FACTOR of 4, giving an external
    -- file record size of 1024-bytes.

    Test_File : TEST_IO.FILE_TYPE;
    -- This is the internal file "handle"
    -- for the external file.

    Test_Data : DATA_RECORD;
    -- This is the 256-byte test data record.

begin -- TEST_DRIVER

    -- Initialize the test data array to
    -- sequential floating-point values.

    INITIALIZE:
    for I in 1..64 loop
        Test_Data := FLOAT(I);
    end loop INITIALIZE;

    -- Now, create (and open) a file
    -- to contain the test data

    TEST_IO.CREATE(FILE => Test_File,
                   NAME  => "TEST.DAT");

    -- Write ten 256-byte records.
    -- We will simply write the same
    -- record ten times.
```

```
-- Since the blocking factor is 4,
-- only ((10 / 4) + 1) external
-- records of 1024 bytes should
-- be written.
```

```
WRITE_DATA_RECORDS:
for I in 1..10 loop
    TEST_IO.WRITE(FILE => Test_File,
                  ITEM  => Test_Data);
end loop WRITE_DATA_RECORDS;
```

```
-- Now, close the external file.
```

```
TEST_IO.CLOSE(FILE => Test_File);
```

```
end TEST_DRIVER;
```

Note how the test program simply indicates that ten data records are to be written to the file. Blocking of the data records is automatically performed by the instantiated package, relieving the user program from having to perform these operations.

Also, since the last external file record contained only two logical records, the remainder of the record was automatically set to zeros when the internal buffer was created.

3 Implementation

3.1 The Underlying Data Structure

The implementation of the Blocked_IO package is based on usage of standard Ada elements. The resources of the Direct IO package are used to implement all of the input-output operations. The data structure used to implement the blocked file can therefore be mapped directly to the underlying direct file records.

The blocked file is an array of ELEMENT_TYPE records. The blocked type, BUFFER.RECORDS, is declared in the package private sections, along with the RECORD_INDEX_TYPE array index type. The operations of Direct_IO are instantiated in the private part of the package specification as the BIO package with the BUFFER.RECORDS type, and the Blocked_IO package FILE_TYPE is defined as a record with its single component Data File as the instantiated package file.type, as shown in below

```
private -- Blocked_IO package

    subtype RECORD_INDEX_TYPE is
        NATURAL range 1..BLOCKING_FACTOR;
    -- Internal record index
```

```

type BUFFER_RECORDS is
  array (RECORD_INDEX_TYPE) of ELEMENT_TYPE;
-- External record data typ

package BIO is
  new DIRECT_IO
  (ELEMENT_TYPE => BUFFER_RECORDS);
-- Instantiation of DIRECT_IO with blocked records

type FILE_TYPE is record
  Data_File : BIO.FILE_TYPE;
end record;
-- External file "handle"

```

3.2 Implementation Design

Implementation of the Blocked_IO package requires an internal "blocked" buffer, three pointers, and a flag variable. Thus, the package functions as a state machine, since it must retain knowledge of its previous state from call to call. These "state" variables are shown below

```

Buffer                : BUFFER_RECORDS;
-- Internal buffer

Current_Block_Number  : COUNT := 0;
-- Resident block number

Current_Record_Number : COUNT := 0;
-- Logical record number

Buffer_Record_Number  : RECORD_INDEX_TYPE;
-- Physical record number

Current_Block_Modified : BOOLEAN := FALSE;
-- "Dirty" block flag

```

The **Current_Block_Number** maintains the number of the external file record ("block"). It is initialized to zero, indicating that no external file record has been read into memory. The **Current_Record_Number** maintains the logical record number, regardless of which external ("block") file record it is in. It is also initialized to zero, indicating no logical record. The **Buffer_Record_Number** maintains the number of the logical record within the current buffer. Finally, the **Current_Block_Modified** flag indicates whether any of the records contained within the current block have been modified. If so, the current block must be written to the external disk file before any other record not in the current block is read or written.

In addition to the functions and procedures identified in the Blocked IO package specification, two additional functions were written in the package body to support the determination of the correct external record number and the

internal record number. The **Is_Block** function takes the logical record number and returns the external file record number. The **Is_Record** function takes the logical record number and returns the number of the record in the external record buffer. Thus, by using these two functions, we can determine which external file record to read in from disk, and which record to access from the internal buffer to obtain the specified logical record.

Calculating the External File Record Number

Calculation of the number of the external file record which contains the specified logical record is performed using integer arithmetic. The logical record number is divided by the blocking factor, and one added to the result. If the logical record is a multiple of the blocking factor, then the external file record number is reduced by one.

Calculating the Internal Buffer Record Number

Calculation of the number of the record within the internal buffer is also performed using integer arithmetic. The result is computed as the logical record number mod the blocking factor. If the result is zero, then the internal record number is set to the blocking factor (the last record in the internal buffer).

Reading Records

The **Read** procedure determines the correct external file record number and internal record numbers, then reads the external record from disk into the buffer contained in the package body. Before reading in the next buffer, however, the procedure first checks to see if the current buffer has been modified. If it has been modified (that is, the "dirty" flag is set), then it writes out the current buffer by calling the **Flush** procedure.

The **Read** procedure is overloaded to allow the user to read records randomly or sequentially. If the record number is not specified, the next logical record is read and returned to the user. The sequential read procedure adds one to the current logical record number, and calls the overloaded direct read procedure.

Writing Records

The **Write** procedure is also overloaded to permit the user to write records randomly or sequentially. If the number of the record to read is not specified, the logical record number is incremented and used to call the overloaded direct write procedure.

The direct **Write** procedure determines the correct external record and internal record numbers, then writes the record to the correct position within the internal buffer. If the external record to write is not the one currently in memory, then the current block buffer is written out, a new buffer is initialized, and the new record written to the new current buffer.

Buffer Initialization

Before writing any record to a new block (that is, a block which will be written after the end of the current file size), a new block buffer is initialized to nulls (zeros). Since the record is furnished at instantiation time, it is not possible to write null records to the buffer unless we required the user to furnish a null record. We chose not to do this. Instead, we overlay a null buffer on top of the internal buffer, as shown in the following code:

```
INITIALIZE: declare

-- First, find out how many bytes in the blocked
-- buffer by obtaining the number of bits in the
-- buffer type and dividing by the number of bits
-- in the SYSTEM_STORAGE unit.

    Buffer_Size : constant POSITIVE :=
        (BUFFER_RECORDS'size /
         SYSTEM_STORAGE_UNIT);

-- Next, create an array with the same number of
-- SYSTEM_STORAGE units, filled with nulls.

    Null_Buffer : array(1..Buffer_Size)
                   of CHARACTER :=
        (others => ASCII.Null);

-- Now, equivalence this null array
-- with the input blocked buffer.

    for Null_Buffer use at Buffer'address;

begin -- INITIALIZE

-- All operations performed in declare statements
null;

end INITIALIZE;
```

Initialization of the internal buffer could have been accomplished by using `UNCHECKED_CONVERSION` to convert the `NULL_BUFFER` to a `BUFFER_RECORDS` type, and copying it to the internal buffer. We decided not to implement this method, as it would require additional memory for the null buffer.

4 Portability Issues

To achieve the goal of portability, only procedures and operations available in Ada were used to implement the package. That is, no DEC VAX system service calls were made in any of the procedures or functions.

One difficulty encountered in the creation of the package was the implementation of the `Form` function. This function returns a string which contains the form information used to open or create the file.

One way to implement this function would be to return the form string obtained from a call to the underlying Direct IO `Form` function, as shown in the following function:

```
function FORM(FILE : in FILE_TYPE)
    return STRING is
begin
    return BIO.FORM(FILE => FILE.Data_File);
end FORM;
```

Unfortunately, this did not work on the DEC VAX. DEC implements the form string using their proprietary File Description Language (FDL). The string returned exceeded the constraint for the return `STRING` type. After a few calls to DEC Customer Support, the problem was resolved, and the function implemented as follows:

```
function FORM(FILE : in FILE_TYPE)
    return STRING is
    Form_String : constant STRING :=
        BIO.FORM(FILE => FILE.Data_File);
begin
    return Form_String;
end FORM;
```

This implementation permits the unusually long form string returned by the underlying Direct IO `Form` function to be assigned to a constant string each time the `Blocked_IO` `Form` function is called. Then, this constant string is returned to the caller.

5 Performance and Tuning

Performance of the `Blocked_IO` package depends upon the size of the logical data record and the blocking factor. The smaller the size of the data record and the smaller the blocking factor, the larger the disk access time. By blocking the data, more data can be written at a time, and fewer disk accesses are required. While larger external file record sizes (data blocks) will result in fewer disk access to transfer data, they will require more memory size to retain the internal buffer. Choice of the blocking factor is, then, a trade-off between how much memory is available and the desired data transfer rate.

Two benchmark programs were written to obtain the time required to write fifty-thousand (50,000) two-hundred fifty-six (256) byte records to disk. The first program called the `Direct_IO Write` procedure to sequentially write the records. The second program called the `Blocked_IO Write`

procedure to write the records sequentially. It was estimated that the Blocked_IO package, instantiated with the 256-byte record and a BLOCKING_FACTOR of ten (10) would execute quicker than the corresponding operations in the Direct_IO package.

In fact, the opposite was found to be true, as shown in the following table:

	Direct_IO	Blocked_IO
Elapsed Time	2:07.56	2:25.47
CPU Time (milliseconds)	14870	27260
Buffered IO (operations)	699	699
Direct IO (operations)	3496	3493

Suspecting a flaw in the procedures, we ran the program with the DEC VAX Symbolic Debugger. Unable to detect any obvious error, we wrote a third test program. This time, we declared Blocked IO as an internal, non-generic package inside the test program. When we ran this pro-

gram with the same test data, we found that the CPU execution time was 12260 milliseconds. This represented a twelve-percent reduction in the Direct_IO execution time, and a fifty-two percent reduction in the execution time of the Blocked_IO procedure.

Since the only obvious difference in the last test program was the lack of generic instantiation of the package, we have assumed that something in the implementation of the instantiation of a generic package is affecting the execution time.

This assumption is supported by the independent article of [Herr 88] which indicates "the inability of many validated Ada compilers to properly handle generic units."

6 Conclusions

The Blocked IO package is an example of how Ada can be extended by means of packages. With care, the user goals of portability and reusability can be met while supporting the Software Engineering goal of understandability. The user goal of speed, however, sometimes eludes even the earnest programmer.

Until Ada compilers correctly and efficiently implement generic packages programmers are denied the primary facility which promotes the creation of reusable software.

7 Blocked_IO Package Specification

```
-----
-----
--
--
-- BLOCKED_IO PACKAGE SPECIFICATION
--

    renames IO_EXCEPTIONS.USE_ERROR;
-- File capacity exceeded, non-existent file.
-- already existing file, or improper FORM string
-- specified during CREATE or OPEN.

    DEVICE_ERROR : exception
        renames IO_EXCEPTIONS.DEVICE_ERROR;
-- Underlying hardware malfunction.

    END_ERROR    : exception
        renames IO_EXCEPTIONS.END_ERROR;
-- Attempt to read record after last
-- block in external file.

    DATA_ERROR  : exception
        renames IO_EXCEPTIONS.DATA_ERROR;
-- Unable to read Blocked Buffer.
-- Possibly wrong blocking factor.

-----
--- BLOCKED_IO PACKAGE PRIVATE SECTION
-----

private

    subtype RECORD_INDEX_TYPE is
        NATURAL range 1..BLOCKING_FACTOR;

    type BUFFER_RECORDS is
        array(RECORD_INDEX_TYPE) of ELEMENT_TYPE;

    package BIO is
        new DIRECT_IO(ELEMENT_TYPE => BUFFER_RECORDS);

    type FILE_TYPE is record
data_file : BIO.FILE_TYPE;

    ITEM :    out ELEMENT_TYPE;
    FROM : in  POSITIVE_COUNT);
-- Returns the element "ITEM" from the logical record
-- number "FROM". Raises END_ERROR exception if "FROM"
-- causes read beyond SIZE(FILE).
```

```

    procedure READ (FILE : in out FILE_TYPE;
    ITEM : out ELEMENT_TYPE);
-- Returns the element "ITEM" from the current logical
-- record number then increments the logical record number.
-- Raises END_ERROR exception if current record number
-- causes read beyond SIZE(FILE).

    procedure WRITE(FILE : in out FILE_TYPE;
    ITEM : in ELEMENT_TYPE;
    TO : in POSITIVE_COUNT);
-- Writes "ITEM" to the logical record number "TO".

    procedure WRITE(FILE : in out FILE_TYPE;
    ITEM : in ELEMENT_TYPE);
-- Increments the current logical record number,
-- then writes "ITEM" to the "FILE".

-- Exceptions

    STATUS_ERROR : exception
        renames IO_EXCEPTIONS.STATUS_ERROR;
-- Attempt to operate on unopened file,
-- or open an already opened file.

    NAME_ERROR : exception
        renames IO_EXCEPTIONS.NAME_ERROR;
-- Invalid "NAME" string specified during
-- CREATE or OPEN.

    USE_ERROR : exception

    function FORM (FILE : in FILE_TYPE) return STRING;
-- Returns the form used to open or create the external file.

    procedure PRINT_FORM (FILE : in FILE_TYPE);
-- Prints the form string associated with the external file.

    function Physical_Record(FILE : in FILE_TYPE)
        return POSITIVE_COUNT;
-- Returns the current (physical) Blocked Buffer record number.

    function Logical_Record(FILE : in FILE_TYPE)
        return POSITIVE_COUNT;
-- Returns the current (logical) record number.

    function Resident_Record(FILE : in FILE_TYPE)
        return POSITIVE_COUNT;
-- Returns the current logical record number
-- (in the current physical record)

    function SIZE (FILE : in FILE_TYPE) return COUNT;
-- Returns the number of the last physical record
-- in the external file.

    function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;
-- Returns TRUE if "FILE" is associated with an external file.

```

```

function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
-- Returns TRUE if "FILE" is at the last physical record
-- in the external file.

```

```

-----
-- Input and output operations
-----

```

```

procedure READ (FILE : in out FILE_TYPE;

procedure OPEN(FILE : in out FILE_TYPE;
NAME : in STRING := "";
FORM : in STRING := "");

-- Associates the file object "FILE" with the
-- external file specified by the name "NAME".

```

```

procedure CLOSE(FILE : in out FILE_TYPE);

-- Disconnects the current association of the
-- file object "FILE" with its external file.

```

```

procedure DELETE(FILE : in out FILE_TYPE);

-- Disconnects the current association of the
-- file object "FILE" with its external file.
-- The external file ceases to exist.

```

```

procedure RESET(FILE : in out FILE_TYPE);

-- Rewinds the file object associated with "FILE".
-- Sets the Block and Record numbers to one.

```

```

procedure FLUSH(FILE : in out FILE_TYPE);

-- Forces the current Blocked Buffer to be
-- written to the file object associated with "FILE".

```

```

-----
-- File information operations
-----

```

```

function NAME (FILE : in FILE_TYPE) return STRING;
-- Returns the name which was used to open or
-- create the external file.

```

```

--
-- BLOCKED_IO PACKAGE SPECIFICATION
--
-- Author: John J. Cupak Jr., CCP
--
-- With Support From : Roy Hoffman
--
-- Date : 24 July 1987
--
-----

```



```

with DIRECT_IO;
with IO_EXCEPTIONS;

generic

    type ELEMENT_TYPE is private;
    -- Logical record type

    BLOCKING_FACTOR : in POSITIVE;
    -- Number of logical records per block

package BLOCKED_IO is

    type FILE_TYPE is limited private;

    type COUNT is range 0..INTEGER'LAST;
    subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;

    procedure CREATE(FILE : in out FILE_TYPE;
        NAME : in STRING := "";
        FORM : in STRING := "");

    -- Creates an external file with the name "NAME"
    -- and associates the file object "FILE" with it.

    type BUFFER_RECORDS is
        array(RECORD_INDEX_TYPE) of ELEMENT_TYPE;

    package BIO is
        new DIRECT_IO(ELEMENT_TYPE => BUFFER_RECORDS);

    type FILE_TYPE is record
        data_file : BIO.FILE_TYPE;
    end record;

end BLOCKED_IO;

```

References

- [Herr 88] Herr, C. S., McNicholl, D. G., and Cohen, S. G. "Compiler Validation and Reusable Ada Parts for Real-Time Embedded Applications," *ACM Ada Letters*, Vol. VIII, No. 3, pp. 75-86, September/October 1988.
- [LRM 83] American National Standard Reference Manual for the Ada Programming Language. ANSI/MIL-STD 1815A-1983. American National Standards Institute, New York, 1983.



NAME: John J. Cupak
 TITLE: Principal Engineer
 EDUCATION: B.A. in Biology, Math co-major
 Syracuse University
 M.S. in Computer Science
 S.U.N.Y at Albany

EXPERIENCE:

Mr. Cupak joined HRB-Singer in September of 1980 and is currently responsible for coordinating activities of the HRB Ada Technical Working Group, supporting the Software Productivity Enhancement Program (S/PEP) and Tools committee, providing guidance and direction to the Ada project staff on design and implementation in Ada. In addition, he is responsible for supporting Marketing and RFP activities.

Mr. Cupak was the RI for the Mission Generation CPCI on project GECE. He was responsible for SRS, design and development of all Ada software for his CPCI. He directed the Ada/PDL selection study for project GECE and assisted in the development of the Ada Programmer's Guide. He also developed the Ada training course for the project.

Mr. Cupak was the lead software engineer for the Generic Electronic Combat Environment Simulation proposal, which was the first Ada proposal won by HRB-Singer.

For a number of years, Mr. Cupak was the RI for the DEOB and directed the design and implementation of emitter location clustering analysis, B-tree, ISAM, and performance monitoring software. Mr. Cupak was also assigned to a special project in which he developed a disk I/O timing analyzer in FORTRAN-77 for the SEL-32/27 under MPX-1.4. Mr. Cupak also developed a prototype Automated Configuration Management System (ACMS) in C and shell.

Mr. Cupak was previously employed by

Pattern Analysis & Recognition, Inc. (PAR technologies). Mr. Cupak was a senior analyst responsible for the design and development of the Remote Data Entry Subsystem for OLPAKS on the Honeywell/MULTICS system using PL/I.

Mr. Cupak was also employed by General Dynamics - Eastern Data Systems Center. As a senior programmer, Mr. Cupak generated and maintained UNIVAC-1100 system software. He implemented a computer job summary account program in structured COBOL to support corporate billing system. He designed Data-graphics 4500 Computer Output Microfilm (COM) control and formatted programs. He designed and implemented the first on-line documentation manual and help commands for proprietary data retrieval language.

Mr. Cupak was a graduate teaching assistant in the Department of Computer Science at the State University of New York at Albany. Mr. Cupak was responsible for developing and teaching structured programming in COBOL. Mr. Cupak's thesis, "Application of List Processing to the Detection of Arterial Lesions by a Labeled Cell Index", applied an innovative single-pass algorithm to detect and recognize objects for subsequent detailed analysis. During a summer at graduate school, Mr. Cupak was a private consultant for Schenectady County Community College, where he designed and implemented a student information system in COBOL.

Mr. Cupak was a programmer/analyst for the New York State Department of Environmental Conservation (DeCon) Administrative Services, where he designed and implemented the Pesticide Reporting System using COBOL and MARK-IV. In addition, he was responsible for supporting small game and turkey wildlife management analysis programs.

Mr. Cupak developed the Ada Certificate program at the Pennsylvania State University and has taught Software Engineering, Introduction to Ada, and Advanced Ada. He was instrumental in causing Data General to donate a MV-1000 Ada Development Environment to Penn State in support of the Ada courses.

During his career, Mr. Cupak has authored/published the following papers:

1. "Application of List Processing to Detection of Arterial Lesions by a Labeled Cell Index", Master's Thesis, SUNYA, 1975
2. "MULTICS Remote Data Entry System", RADC-TR-79-265
3. Co-author - "B-Trees for Directory Organization"
4. Internal project study - "Ada/PDL Comparisons", 1984, "Ada Course", 1986, and "OOD with Ada", 1987.

Mr. Cupak belongs to the Association for Computing Machinery (ACM), the ACM SIGAda, and the ICCP societies. Mr. Cupak is the chairperson for the HRB-Singer Ada Technical Working Group and is a member of the Singer Corporate Ada Working Group.

DEVELOPING A UNIVERSAL Ada TEST LANGUAGE for GENERATING SOFTWARE/SYSTEM INTEGRATION and FAULT ISOLATION TEST PROGRAMS

by Jehuda Ziegler, Jerry M. Grasso, Linda G. Burgermeister, Leonard D. Molloy

ITT Avionics, 390 Washington Avenue, Nutley, NJ 07110-3603 (201) 284-2030

ABSTRACT

ITT Avionics has developed a Universal Ada Test Language (UATL) for writing automated test programs to perform extensive real time software/system performance testing and factory production/field maintenance fault detection/isolation. The UATL was developed as a STARS (Software Technology for Adaptable Reliable Systems) Foundations project to provide support to the replacement of ATLAS, BASIC, and other special purpose test control languages with Ada. The UATL consists of a set of portable Ada packages that provide the user with a complete complement of standardized reusable test functions. These functions include an interactive menu driven test manager, on-line operator controls/displays, real-time "closed loop" test data stimulus/response, instrument control drivers, test data recording, and both ASCII and graphical data reduction analysis. Currently, the UATL supports driving a software unit-under-test (UUT) over internal memory, or a hardware UUT with a set of stimulus and measurement instruments over IEEE 488 and MIL-STD-1553B interfaces. It has been designed to readily support the addition of more reusable test control or analysis functions to the UATL "test language" library, and allows the user to develop any unique test functions at the Ada code level.

INTRODUCTION

DoD directive 3405.2 [1] specifies Ada as the *preferred* language for generating automated test programs. It is not required at present because of the standardized test support functions currently available in ATLAS. However, the long range goal is to transition all DoD software to Ada [2].

The major strength of the ATLAS language has been its UUT test signal oriented vocabulary that can be used to specify tests that are independent of the specific test station in use [3]. The UATL instrument control procedures have been modeled after the ATLAS constructs to retain familiarity with this IEEE/ANSI standard test language [4] and make the transition to Ada relatively straightforward and cost effective.

However, although ATLAS can support hardware fault detection/isolation testing, it cannot be used to perform real time software/system integration testing. ATLAS is limited in its support of digital testing, does not provide the real time high data rate throughput that is necessary for integration testing, and does not contain any constructs for data recording and reduction capabilities needed for extensive software/system performance analysis.

Software/system development, acceptance, and maintenance (regression) testing has traditionally been performed with special purpose test equipment controlled by non-standard test scenario languages. The DoD Computer Programming Language Policy directives [1,2] do not address the issue of standardizing integration test control languages.

The UATL has been designed as a general multi-purpose test language that provides a consistent framework for testing at all phases of the software/system development, production, and maintenance life cycle. It supports extensive real time software integration testing at the CSC/CSCI level within a host development processor over a tasking mailbox interface (Fig. 1), and hardware/software integration testing or hardware fault detection/isolation of a complete system/subsystem over MIL-STD-1553B and IEEE 488 interfaces (Fig. 2).

Ada introduces the advantages of built-in multi-tasking support, extensive data consistency checking, portability and reusability of recurring test functions, and compiled code that runs much faster than ATLAS.

By supporting extensive software integration testing in the host processor software development environment the UATL encourages the rapid prototyping and validation of system functions early in the development process. This greatly reduces the number of untested functions that must be verified on the final system hardware where test resources are usually very limited.

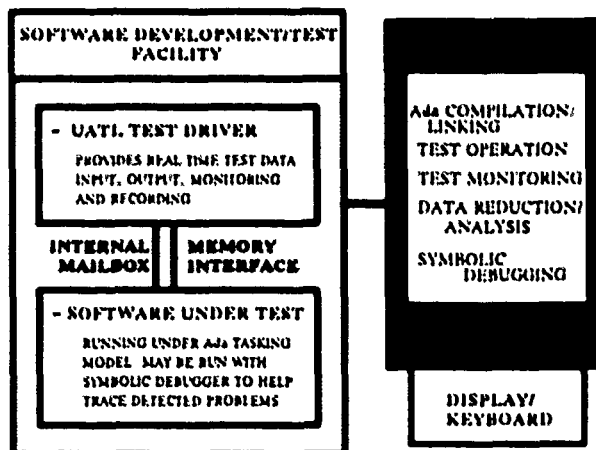


Fig. 1. Application to software CSC/CSCI testing within a standalone host processor software development facility.

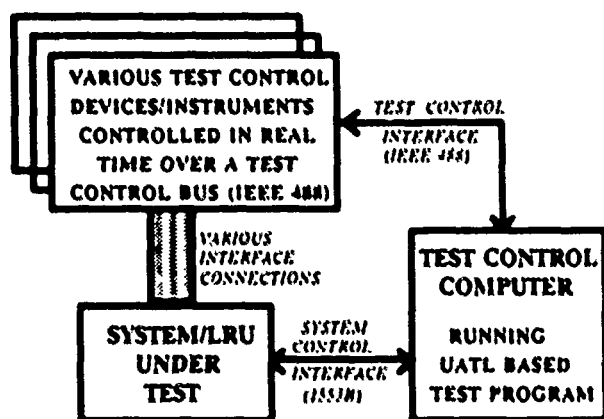


Fig. 2. Application to hardware/software integration and factory/maintenance fault isolation testing over the system control bus and via digital control of various test devices/instruments over the test control bus.

Additional UATL components have also been developed to aid test program generation, provide on-line operator test control and monitoring functions, and post-test data reduction and analysis. Future extensions to support data IO over additional interfaces, control additional test instruments, or accomplish other reusable test functions are easily added to the UATL test support library.

As part of the STARS project requirements, portability of the UATL software has been demonstrated by operating both on a VAXstation II under VMS and on an IBM AT compatible (ITT XTRA/286) under MS-DOS. Portability has been designed into the UATL by limiting any unique hardware dependent software to configuration dependent package bodies, "separate" procedures, or initialization data files.

In reality it turns out that converting ATLAS test programs to run on different test stations is not a trivial process and it will be just as cost effective to convert them to Ada which is much more supportive of portability. An automated ATLAS to UATL/Ada conversion tool can also be developed to retain the existing large investment in ATLAS test programs.

The UATL received the "Most Innovative Product" award at the STARS Current Products and Future Directions Workshop held Nov. 28 - Dec. 2, 1988 in Arlington, VA.

ADVANTAGES IN USING Ada

Current arguments for the support of ATLAS [5,6] are related to the availability of readable higher level test (station independent) commands in ATLAS. With the development of the UATL and its reusable, standardized Ada test procedures labeled with the long readable names supported by Ada, most of these arguments are removed.

There are also many advantages for using Ada as the standard test control language. In most current implementations, the fault detection and isolation programs for on-line BIT, factory production test, and for organizational, intermediate, and depot level maintenance, are rewritten independently in various languages to run on the actual system hardware or the different test stations. By

standardizing on the use of Ada for all the test software, these multiplicative efforts will be eliminated resulting in both non-recurring test program development and recurring maintenance cost savings. Ada supports the development of portable application test program packages that detect and isolate faults in the UUT to the LRU/SRU (line/shop replaceable unit) and component level. The appropriate test packages are then "with"ed in and used as required by the test programs for each target test station.

The UATL was designed to provide as many reusable test functions as possible to reduce the test generation effort, but it does not restrict the user from generating any unique test controls at the Ada code level. Since ATLAS is an interpreted language, it restricts the user to what is defined in the language and what is supported by the ATLAS compiler and run time executive (interpreter) in each specific test station. When instruments with new functions become available many current ATLAS implementations require the user to import compiled code (such as FORTRAN) to accomplish the required test tasks.

The strong Ada data type checking can also be used to guarantee that the test data is consistent with the UUT by "with"ing in the actual Ada interface data specifications from the software-under-test in the UATL based test programs, the Ada compiler will automatically verify that the test data is consistent.

The faster operation (about an order of magnitude in many cases) of compiled Ada code compared to interpreted ATLAS, will considerably reduce system MTTR (mean time to repair) with its resultant recurrent time and cost reductions. The Ada multi-tasking capability also supports the running of parallel digital and analog (when the instruments are available) test functions which also improve the test throughput for a given set of test station resources. The performance tests written in ATLAS for intermediate level maintenance testing of some of the more complex LRUs on a major project at ITT Avionics, take several hours to run, and tens of hours to link, on the Advanced Electronic Warfare Test Set (AEWTS). There is thus much room for improvement in this area.

The design of the UATL has also been driven by the STARS requirement for portability. The UATL software was developed on a VAXstation II and ported to an ITT XTRA/286 (IBM AT compatible). The Ada specifications and all the higher level code are common to both the VAX and XTRA implementations. The unique software that was required to interface with and control the different target hardware was limited to configuration dependent package bodies and "separate" procedures.

Ada truly lives up to its portability claims. Less than 3% of the code had to be modified in order to port to the ITT XTRA, mainly in order to interface with the different 1553 and IEEE 488 interface hardware. The only restrictions that we found on rehosting the UATL software to the ITT XTRA were in the memory management limitations of the MS-DOS operating system and in some of the optional (chapter 13) Ada language functions that were not tested by the Ada validation suite. These problems will disappear when the chapter 13 functions are tested and with a more robust operating system such as UNIX or PS/2.

Ada's portability will enable the UATL to be easily rehosted to newer, faster processors as required to meet test requirements. The UATL will thus never become restricted by obsolete processors that may have been built into test stations such as is currently the case with ATLAS.

We have found that Ada also greatly improves the quality and productivity of the software engineering process. The entire UATL effort consisting of the design, code, and integration testing of 1,044,560 bytes of generated object code, including the recoding of 24,012 bytes for the rehosting to the ITT XTRA, was completed and demonstrated in a 12 month period by a team of 5 software engineers.

In addition, many CASE (computer aided software engineering) tools are commercially available, or being developed, that support software development and maintenance in Ada.

BASIC UATL FUNCTIONS

UATL test support functions are provided for Test Program Generation, On-line Test Operation, Real-time Test Control, and Post-Test Data Reduction. The interactive menu driven Test Manager helps manage the test station and test program configuration and provides menu driven controls for creating and running test and data reduction programs. The Control/Display supports On-line Test Operation, and the Trajectory Computation functions are used to support both On-line Test Operation and Post-test Data Reduction analysis.

The UATL has been designed with a building block, layered, approach. Each reusable test function is provided with a standardized Ada interface specification to the next higher level. The set of Ada packages for each function was also separated into common, portable, compilation units and configuration/hardware dependent package "bodies" and "separate" procedures. Where practical, the UATL components were designed to be configurable via ASCII configuration files that are read during test initialization. This allows the maximum portability of the UATL to various test station configurations.

As illustrated in Fig 3, the Stimulus/Response, Control/Display, Data Recording, Data Reduction, and IEEE 488, MIL-STD-1553B, and Internal Memory Digital I/O CSCs provide a complete digital test capability and form the baseline to which the additional test functions are added. The Stimulus/Response, Digital IO, and Data Recording functions support the real time running of the automated test programs. The Control/Display function provides on-line operator test controls, and the Data Reduction function provides a detailed off-line analysis capability.

Digital I/O Function

The Digital IO function provides the ability to interface a UATL based test program to the UUT over various interfaces. Currently (Fig. 3) three packages are provided with procedures/tasks for sending and receiving messages via the MIL-STD-1553B and IEEE 488 buses, or via an internal memory mailbox interface. This can easily be expanded to add packages for controlling I/O over other interfaces such as the new standard VXLbus for instruments.

The Digital IO CSCs have been designed to insure maximum portability by limiting the hardware specific code to configuration dependent package bodies and "separate" procedures. This has been demonstrated in the rehosting of these packages from the MicroVAX and its Q-bus based interface cards to the ITT XTRA with its PC-bus.

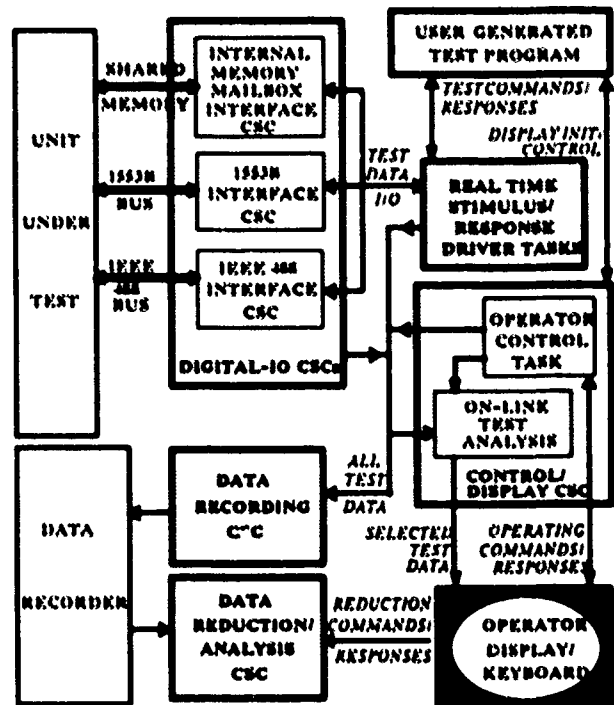


Fig. 3. Basic UATL components and their interfaces. The design supports the addition of packages to handle digital IO over additional interfaces, control various test instruments, perform data analysis, etc.

It should be noted that these CSCs have been designed to be of general utility and can be used as standalone reusable interface drivers by any Ada software to interface with the IEEE 488 or MIL-STD-1553B buses or to other tasks within the same processor via mailbox messages.

Data Recording Function

The Data Recording CSC provides procedures for on-line recording of all, or selected, test data on disk or magnetic tape cartridge for use in detailed off-line data reduction analysis. This function is necessary in order to allow the test program to run in real time and collect the data for detailed post test analysis. The only analysis that is required to be accomplished in real time is what is needed to interactively change the test operation.

Stimulus Response Function

The Stimulus Response CSC provides the capability to: send stimulus messages at a specified time and rate with the ability to vary the value of selected field(s) on each output, save and compare output responses from the UUT and

determine if they are received within specified response timeouts, modify the test sequence by activating/deactivating events based on received trigger responses, and detect faults based on the reception of incorrect or non-reception of expected data.

As illustrated in Fig. 4, the UATL provides three basic events for controlling test operation: stimulus, response and trigger tasks. Multiple events can be run in parallel under the Ada tasking model. The stimulus task sends messages to the UUT via the Digital IO interface handlers. The response task receives messages from the UUT and calls the comparator tasks as required. The trigger task is used to effect real-time test sequence changes in reaction to messages received from the UUT by activating/deactivating (triggering) other events. Direct control of the UATL events is also provided via procedure calls from the user test program.

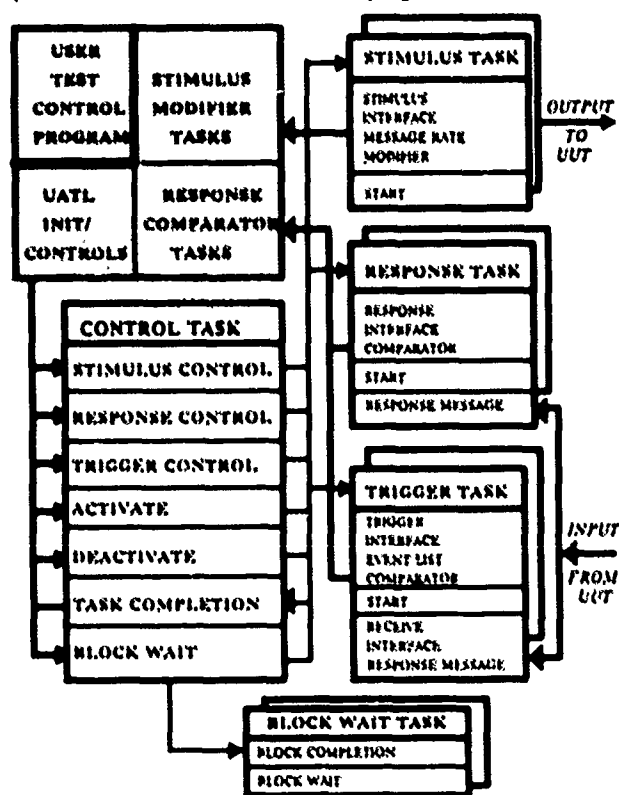


Fig. 4. Stimulus/Response Tasks and their interactions with the user test program and UUT.

The stimulus event is defined by a list of messages and the rate at which they are to be transmitted. Transmission of the entire list can be repeated by the specified circulate count, and restarted by the specified cycle count. A modifier task, which will be called to modify the message before its output, can also be assigned to each message. Each event can be set to run freely without interruption, wait at the first cycle for a trigger, or wait at the start of every cycle for a trigger event.

The response event is specified with a set of storage buffers to save received messages from the UUT. Other buffers can be defined to save the indexes which point to the saved error responses, the error response receive count. The error

status of a particular response is determined by the user supplied comparator tasks. A "no response" timeout error condition can also be determined if no match is found for the expected messages. The response event also has the capability to interrogate devices on the interface to elicit a response controlled via the response delay timing parameter.

The trigger event is defined as a list of stimulus, response, or other trigger tasks that are to be activated or deactivated when the trigger occurs. A trigger occurrence is determined by a match condition returned by the associated comparator task. Each trigger task can also be set to sequence through a trigger list each time the current trigger is satisfied by a "match".

The block wait function is also available to halt execution of the "main" test procedure until the group of events with a specified block name are completed. Block completion can be specified as the completion of all, or just the first, event in the block. The UATL will shut down all events in a block before returning control to the test procedure.

Control/Display Function

The Control/Display function provides the operator with the ability to monitor and control the on-line test program execution. The operator can activate, deactivate, and control the operation of the tests, on-line test analysis, and data recording. The control display function also supplies subroutines which support the input/output of messages between the test program and the operator.

The user test program has the option of activating the Control/Display function by calling a supplied procedure. Once this call is executed the initial screen with the main menu is displayed.

Main=> REcording RUStatistics TRaffic MESSage QUIt

The 'RUN' command will start, and 'QUIT' will stop, test execution. The 'REcording' option allows the user to activate/terminate data recording or modify the list of messages (identified by Batch ID) that will be recorded.

The 'Statistics' option provides a real time monitor display of overall test execution. As illustrated in Fig. 5, this displays how many messages for each batch ID have been exchanged over each interface. The statistics display has a menu which allows the operator to scroll through all the selected batch ID's, change the list of selected messages to be counted, freeze the screen update, go to the TRaffic display, or go to the MESSage Display.

The 'TRaffic' option provides the real time display of the current message traffic being exchanged with the UUT. As shown in Fig. 6 the last five messages exchanged in each direction are displayed at any given time. The operator has the option to select which messages will be displayed, freeze the screen, go to the Statistics display, or go to the MESSage display.

Message Statistics				
Batch ID	IEEE488	INTERNAL	LOCAL	TOTALS
0			1	1
1			11	11
2		48		48
3		38		38
4				0
5			88	88
6				0
7				0
8				0
9		63		63
10			14	14
11	237			237
12	25			25
13	120			120
14	75			75
TOTALS 195 262 147 112 716				
Statistics=> Next Previous Selection Traffic Freeze				

Fig. 5. Sample Message Statistics Monitor Display Screen

Message Traffic Data					
Messages to UUT			Messages from UUT		
Time	Batch ID	Interface	Time	Batch ID	Interface
13:46:45:26	20		13:46:47:36	15	INTERNAL
53 0101 0304 0506			19 0204 0606 0A0C		
13:46:46:26	10		13:46:47:58	13	IEEE488
54 0204 0606 0A0C			20 0102 0304 0506		
13:46:47:26	14	IEEE488	13:46:48:27	6	IEEE488
55 0111 0205 1A21			21 0A02 0304 0406		
13:46:48:36	16	INTERNAL	13:46:48:38	11	IEEE488
56 0304 0506			22 0102 0304 047F		
13:46:44:58	12	IEEE488	13:46:47:25	2	IEEE488
52 0203 0455 0555			18 0204 0606 0A0C		
Traffic=> Selection Statistics Freeze					

Fig. 6. Sample Message Traffic Data Monitor Display Screen

The Control Display function also provides procedures which can be called from the user's test program to display messages to, and receive input from, the test operator. The Operator Message and Operator Alert (flashing output) procedures display a message at the bottom of the current control display screen, and the Operator Prompt procedure displays a message and waits for an operator response.

The 'MMessage' option provides a real time display screen for the operator display and prompt messages. The most current messages are displayed on the screen with an arrow pointing to the latest. When the other displays are activated, the operator messages are displayed at the bottom of the screen.

Data Reduction/Analysis Function

Data Reduction provides both ASCII and graphical support functions for analyzing the recorded test data.

For a tally of all the messages in the recorded file, and an indication of whether any messages output to recording did not get recorded, the user calls the Message_Statistics procedure with a control table indicating the IDs of the recording batches to be counted and the output print file

name. This creates the message count statistics print file illustrated in Fig. 7.

MESSAGE STATISTICS PRINTOUT					
Test : WRA 5			Run : SN: 105A		
START TIME : 2/18/1988			10:24:29.30		
Batch ID's	IEEE488	INTERNAL	LOCAL	Totals	
0	0	0	0	1	1
1	25	0	0	0	25
11	0	316	7	0	316
13	35	0	1	0	36
15	0	0	230	0	230
122	0	0	0	79	79
total 60 316 245 80 701					

Error Statistics

Lost sequence numbers		between times
from	to	
2	2	00:00:13.89 - 00:00:52.93
7	15	00:01:10.18 - 00:02:23.45
135	157	00:10:05.36 - 00:12:37.29

Fig. 7. Sample Data Reduction Message Statistics Printout

For a printout of the recorded message contents, the user calls the Message_Printout procedure with a control table indicating the IDs of the recording batches to be printed and the output print file name. If an application dependent reduction task is not provided, then the default printout shown in Fig. 8 of the decoded UATL defined headers and a hex dump of the message data is created. If an application dependent reduction task is provided, then the user determined ASCII strings will be printed for the message data contents following the message header printouts.

MESSAGE DATA PRINTOUT	
Test : WRA 5	
START TIME : 5/25/1988	
Run : SN 105 A	
08:58:12.80	
MESSAGE : Batch ID = 0	Sent at: 00:00:01.87
Sequence Number = 1	Data Byte Count = 26
Interface = LOCAL	
MESSAGE : Batch ID = 14	Sent at: 00:00:03.08
Sequence Number = 3	Data Byte Count = 8
Interface = IEEE488	
RT = 1 TRANSMIT	Subaddr = 17 Word Count=3
0401 0000 0052	
MESSAGE : Batch ID = 20	Sent at: 00:00:04.68
Sequence Number = 4	Data Byte Count = 21
Interface = LOCAL	
UATL EVENT MESSAGE	
Block : SET POWR	Event : HP PWR LVL
Interface : IEEE488	State : RUN Trigger Mode : RUN
MESSAGE : Batch ID = 12	Sent at: 00:00:06.76
Sequence Number = 6	Data Byte Count = 22
Interface = IEEE488	
Receivers = 6	
Message Byte Count = 16	
4C 45 56 45 4C 2D 31 32	
30 2E 30 30 20 20 20 20	

```

MESSAGE :: Batch ID = 1 Sent at: 00:00:09.51
Sequence Number = 8 Data Byte Count = 82
Interface = LOCAL
!!! TEST IS COMPLETE !!!

```

Fig. 8. Sample Data Reduction Message Data Printout.

For general reduction analysis the user calls the Message_Analysis procedure with a control table indicating the IDs of the recording batches to be analyzed, and a user supplied application dependent analysis task. This user task will be passed all the desired recording batches for analysis. The user can output the results his analysis as ASCII strings that will be printed or as plot data to be graphed.

For graphical analysis, the UATL provides procedures to create either bar graph (Fig. 9) or multi-line x-y plots (Fig 10). The UATL procedures will create the graphs, but the user must provide the data to label the graphs and provide the plot data. The graphical output can also be saved in a GKS Metafile.

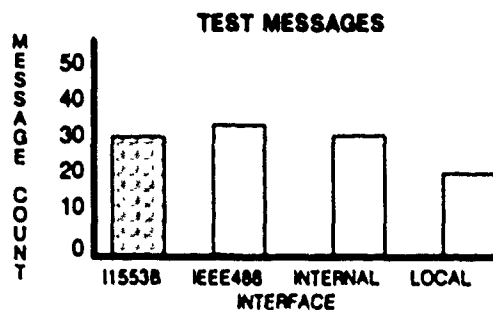


Fig. 9. Sample Bar Graph Output Screen/Printout

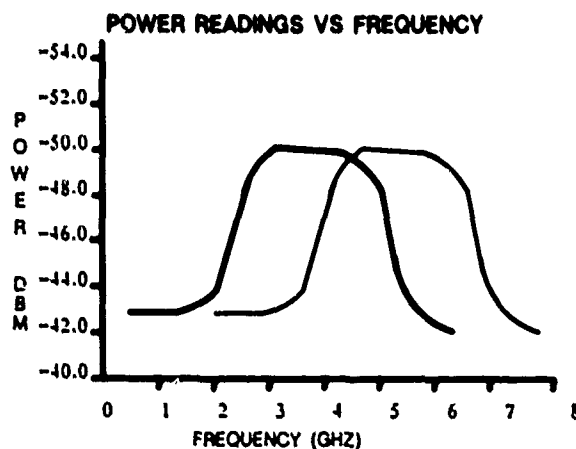


Fig. 10. Sample X,Y Plot Graphical Output Screen/Printout

The generic Bar_Graph package provides procedures that allow the user to create a bar graph. The Initialize_Bar_graph procedure allows the user to specify a title, x, y -axis labels, the number of bars, and whether the bars will be vertical or horizontal. The Display_Bar_Graph procedure allows the user to specify the size, color, and fill shading for each bar.

The generic XY_Plot packages (for integer, float, and fixed data) allows the user to specify a title, x, y -axis labels, and

the number of curves with their color and line type. Each plot point is associated with a selected curve. To assist in the graphical analysis, curve smoothing and least squares fit functions are provided. The trajectory computations package can also be used to perform position accuracy analysis.

INSTRUMENT CONTROL FUNCTIONS

As illustrated in Fig. 11, we have added various test instrument drivers that translate Ada procedure calls into IEEE 488 bus commands to control various test instruments, to the basic UATL functions described above. These instrument drivers are supplied at several levels.

General ATLAS type test commands are supported by the Functional Instrument Control package. This enables the user to write test station independent programs at the UUT level. The Functional Instrument Control package then converts the user commands to specific instrument controls based on test station and UUT interface device configuration data.

Instrument controls are provided through the instrument driver packages. A standard MATE-CIIL (Modular Automated Test Equipment - Control Interface Intermediate Language) instrument driver provides bus commands that will control any instrument that responds to the Air Force MATE standard CIIL commands. Instruments that have additional capabilities that are not included in the CIIL standard can be accessed via the "alternate" language command drivers. Specific instrument control packages are also provided to process commands in an instrument's "native" language. Drivers are also provided to control signal switching devices that are used to route stimulus and measurement data to the appropriate instruments.

The UATL also provides a library of reusable generic test programs that can be used to test standard components of various systems, such as a MIL-STD-1553B interface, an amplifier, etc. This provides test capabilities at a higher level than the ATLAS UUT directed commands and facilitates the development of standardized tests for similar components included in various systems. This also greatly reduces the test program generation effort.

User test programs "with" in these instrument and higher level test control packages to control the devices on the IEEE 488 instrument bus. Similar packages can be developed to control instruments over other test buses, such as the new instrument-on-a-card VXIbus standard, by modifying the package bodies of the lowest level UATL instrument specific drivers. The UATL instrument control specifications, and all test programs developed with them, will not have to be modified. It is expected that if Ada becomes the accepted test language, these types of packages will in the future be provided by the instrument and test station vendors.

Functional Instrument Controls

The Functional Instrument Control package provides test station independent instrument controls to the user. These controls are Ada procedures which provide the functionality of ATLAS statements using the verbs *apply*, *remove*, *measure*, *verify* and *read*. For more specialized instrument controls the user can access the UATL instrument specific packages directly. For reusable, test station and UUT independent

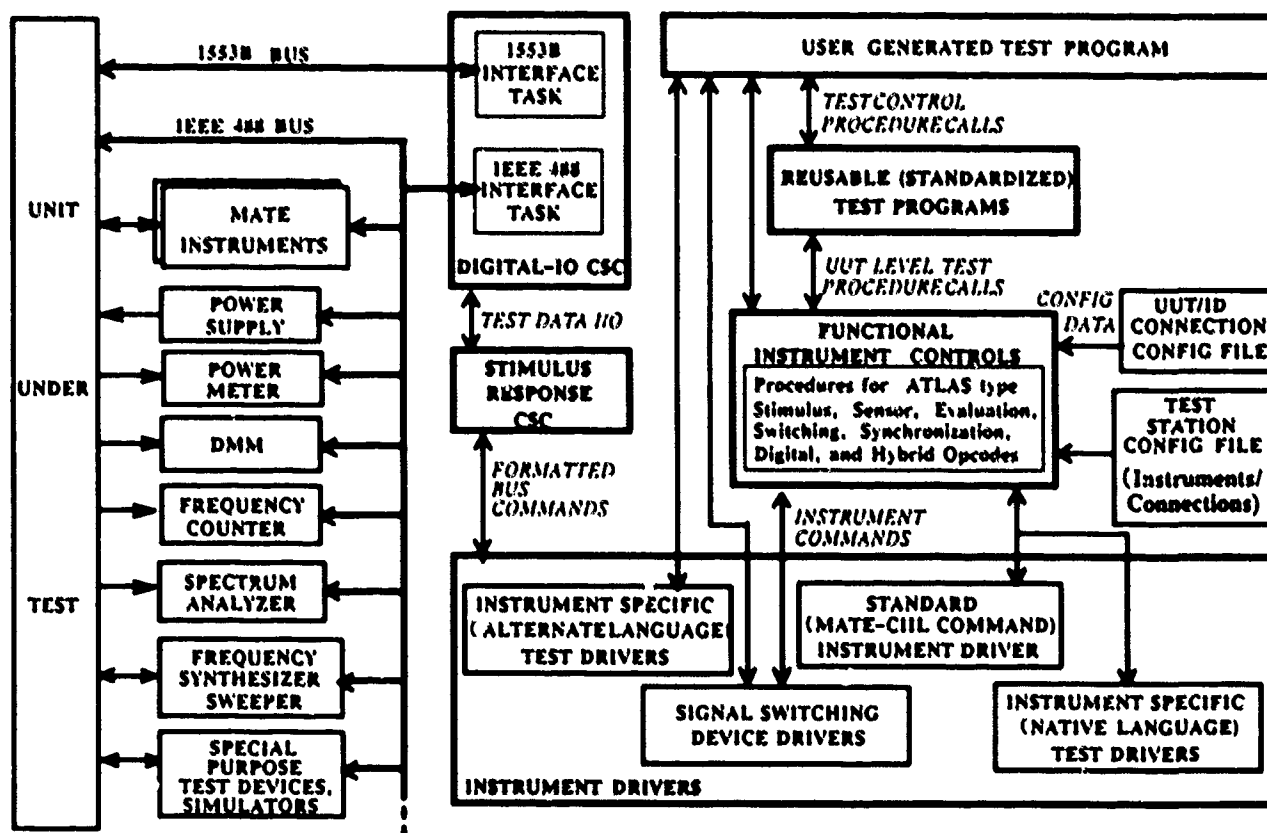


Fig. 4. Addition of the Reusable Standardized Test Programs, Functional Instrument Controls, and Instrument Specific software drivers are provided to control various Instruments via standard CIIL or specific (native or alternate language) instrument commands. The Functional Instrument Control package enables the user to write test programs at the UUT level and converts them to specific instrument commands based on test station and UUT interface device configuration data.

tests, generic test packages can be built up from the functional instrument commands. The functional package provides virtual test station independent controls which are then mapped to the physical instruments through the use of an ASCII test station configuration file. Selected functional instrument control procedures and data types for the physical quantities that are used is provided in Figure 12 below.

Package FUNCTIONAL_INSTRUMENT_CONTROL_TYPES is
-- ...

```

subtype METERS is FLOAT range FLOAT'first..FLOAT'last;
subtype HERTZ is FLOAT range FLOAT'first..FLOAT'last;
subtype SECONDS is FLOAT range 0.0..FLOAT'last;
subtype VOLTS is FLOAT range -10.0E3..+10.0E3;
subtype AMPERES is FLOAT range 0.0..1.0E3;
subtype WATTS is FLOAT range 0.0..10.0E6;
subtype OHMS is FLOAT range 0.0..FLOAT'last;
subtype DBM is FLOAT range -150.0..+200.0;

subtype RF_FREQ is HERTZ range 1.0E3..2.0E9;
subtype MICROWAVE_FREQ is HERTZ range 2.0E9..26.0E9;
subtype LIGHT_FREQ is HERTZ range 4.3E14..7.5E14;

```

```

KILOHERTZ : constant HERTZ := 1.0E3;
MEGAHERTZ : constant HERTZ := 1.0E6;
GIGAHERTZ : constant HERTZ := 1.0E9;

```

```

MILLIMETER : constant METERS := 1.0E-3;
CENTIMETER : constant METERS := 1.0E-2;
KILOMETER : constant METERS := 1.0E3;

NANOSECOND : constant SECONDS := 1.0E-9;
MICROSECOND : constant SECONDS := 1.0E-6;
MILLISECOND : constant SECONDS := 1.0E-3;
MINUTE : constant SECONDS := 60.0;
HOUR : constant SECONDS := 3600.0;
DAY : constant SECONDS := 86400.0;

```

```

MICROWATT : constant WATTS := 1.0E-6;
MILLIWATT : constant WATTS := 1.0E-3;
KILOWATT : constant WATTS := 1.0E3;
MEGAWATT : constant WATTS := 1.0E6;

MICROVOLT : constant VOLTS := 1.0E-6;
MILLIVOLT : constant VOLTS := 1.0E-3;
KILOVOLT : constant VOLTS := 1.0E3;

MICROAMPERE : constant AMPERES := 1.0E-6;
MILLIAMPERE : constant AMPERES := 1.0E-3;

MILLIOHM : constant OHMS := 1.0E-3;
KIOHM : constant OHMS := 1.0E3;
MEGOHM : constant OHMS := 1.0E6;

```

-- ...
end FUNCTIONAL_INSTRUMENT_CONTROL_TYPES;

```

with FUNCTIONAL_INSTRUMENT_CONTROL_TYPES;
use FUNCTIONAL_INSTRUMENT_CONTROL_TYPES;

Package FUNCTIONAL_INSTRUMENT_CONTROL is
-- ...

procedure GET_RESOURCE
  (ADDRESS : in ID_TYPE;
   GRANTED : out BOOLEAN;
   HOLD : in RECEIVE_CALLS := WAIT;
   HOLD_TIME : in DURATION := 0.0);

procedure RELEASE_RESOURCE (ADDRESS : in ID_TYPE);

procedure APPLY_AC_FREQUENCY
  (USING : in LOGICAL_DEVICE;
   FREQUENCY : in HERTZ;
   POWER : in DBM;
   CONNECT_VIA : in STRING := "";
   STATUS : out DRIVER_STATUS);

procedure APPLY_SWEEP_AC_FREQUENCY
  (USING : in LOGICAL_DEVICE;
   START_FREQUENCY : in HERTZ;
   STOP_FREQUENCY : in HERTZ;
   FREQUENCY_STEP : in HERTZ;
   POWER : in DBM;
   STEP_DURATION : in DURATION;
   CONNECT_VIA : in STRING := "";
   STATUS : out DRIVER_STATUS);

procedure MEASURE_AC_FREQUENCY
  (USING : in LOGICAL_DEVICE;
   START_FREQUENCY : in HERTZ;
   STOP_FREQUENCY : in HERTZ;
   MEASURED_FREQUENCY : out HERTZ;
   CONNECT_VIA : in STRING := "";
   STATUS : out DRIVER_STATUS);

procedure MEASURE_AC_POWER
  (USING : in LOGICAL_DEVICE;
   FREQUENCY : in HERTZ;
   POWER : out DBM;
   CONNECT_VIA : in STRING := "";
   STATUS : out DRIVER_STATUS);

procedure MEASURE_AC_VOLTAGE
  (USING : in LOGICAL_DEVICE;
   VOLTAGE : out VOLTS;
   CONNECT_HI : in STRING := "";
   CONNECT_LO : in STRING := "";
   STATUS : out DRIVER_STATUS);

procedure MEASURE_AC_CURRENT
  (USING : in LOGICAL_DEVICE;
   CURRENT : out AMPERES;
   CONNECT_HI : in STRING := "";
   CONNECT_LO : in STRING := "";
   STATUS : out DRIVER_STATUS);

```

-- ...
end FUNCTIONAL_INSTRUMENT_CONTROL;

Fig. 12. Selected Functional Instrument Control data types and control procedures.

Instrument Specific Controls

Fig. 13 is an example of an instrument specific package for controlling a frequency synthesizer. Note the instrument specific data typing that restricts the general physical parameters to those supported by the instrument. Ada provides constraint checking to ensure that the limits are not exceeded during operation.

```

with FUNCTIONAL_INSTRUMENT_CONTROL_TYPES;
use FUNCTIONAL_INSTRUMENT_CONTROL_TYPES;

package GIGATRONICS_1018_FREQUENCY_SYNTHESIZER_TYPES is
-- ...

-- Gigatronics specific ranges
subtype GIGA_FREQUENCY_SWEEP_TYPE is
  HERTZ range 100.0*MEGAHERTZ..18_000.0*MEGAHERTZ;
subtype GIGA_FREQUENCY_SWEEP_STEP_TYPE is
  HERTZ range 1.0*MEGAHERTZ..18_000.0*MEGAHERTZ;
subtype GIGA_FREQUENCY_MEASUREMENT_TYPE is
  HERTZ range 50.0*MEGAHERTZ..18_000.0*MEGAHERTZ;
subtype GIGA_OUTPUT_LEVEL_TYPE is
  DBM range -119.9..15.0;
subtype SWEEP_STEP_DURATION_TYPE is
  DURATION range 0.0001..1000.0;

-- ...
end GIGATRONICS_1018_FREQUENCY_SYNTHESIZER_TYPES;

with FUNCTIONAL_INSTRUMENT_CONTROL_TYPES;
use FUNCTIONAL_INSTRUMENT_CONTROL_TYPES;
with GIGATRONICS_1018_FREQUENCY_SYNTHESIZER_TYPES;
use GIGATRONICS_1018_FREQUENCY_SYNTHESIZER_TYPES;

package GIGATRONICS_1018_FREQUENCY_SYNTHESIZER is
-- ...

procedure GENERATE_FIXED_FREQUENCY
  (BUS_ADR : ID_TYPE;
   FREQUENCY : GIGA_FREQUENCY_SWEEP_TYPE;
   OUTPUT_LEVEL : GIGA_OUTPUT_LEVEL_TYPE;
   STATUS : out DRIVER_STATUS);

procedure SET_POWER_LEVEL
  (BUS_ADR : ID_TYPE;
   OUTPUT_LEVEL : GIGA_OUTPUT_LEVEL_TYPE;
   STATUS : out DRIVER_STATUS);

procedure GENERATE_SWEEP_FREQUENCY
  (BUS_ADR : ID_TYPE;
   START_FREQUENCY : GIGA_FREQUENCY_SWEEP_TYPE;
   STOP_FREQUENCY : GIGA_FREQUENCY_SWEEP_TYPE;
   FREQUENCY_STEP : GIGA_FREQUENCY_SWEEP_STEP_TYPE;
   LOCKED_ON_UNLOCKED : SWEEP_TYPE := LOCKED;
   OUTPUT_LEVEL : GIGA_OUTPUT_LEVEL_TYPE;
   SWEEP_MODE : SWEEP_MODE_TYPE := SINGLE;
   SWEEP_STEP_DURATION : SWEEP_STEP_DURATION_TYPE;
   STATUS : out DRIVER_STATUS);

-- ...
end GIGATRONICS_1018_FREQUENCY_SYNTHESIZER;

```

Fig. 13. Selected instrument specific data types and control procedures for operating a Gigatronics frequency synthesizer.

Figure 14 presents comparisons of ATLAS to UATL functional and direct instrument level test programs. As can be seen, the meaningful variable and procedure names supported by Ada make the code just as, if not more, readable than the mnemonic type ATLAS commands.

```

*****
** ATLAS PROGRAM EXAMPLE:

200710 BEGIN, TEST-BLOCK $
200720 APPLY, AC SIGNAL USING 'RFSRL',
      POWER 10DBM,
      FREQ 10E9HZ,
      CNX VIA J5 $
200730 END,TEST-BLOCK $

```

```

-----
-- UATL FUNCTIONAL INSTRUMENT CONTROL TEST FORMAT:

APPLY_AC_FREQUENCY
(USING      => FREQUENCY_GENERATOR_1.
 FREQUENCY  => 10.0E9.  -- HZ
 POWER      => 10.0.    -- DBM
 CONNECT_VIA => J5,
 STATUS     => STATUS);

-----
-- UATL INSTRUMENT SPECIFIC PROGRAM FORMAT:

J5 SET_SWITCH  -- To connect Frequency_Generator to
(BUS_ADDRESS => 12,
 SETTING     => A1,
 STATUS      => STATUS);

GENERATE_FIXED_FREQUENCY
(BUS_ADR     => 15,
 FREQUENCY   => 10.0E9.  -- HZ
 OUTPUT_LEVEL => 10.0.    -- DBM
 STATUS      => STATUS);

```

Fig. 14. ATLAS to UATL based test program comparisons for applying a fixed frequency, fixed power level, AC signal to UUT J5.

Mapping Of Functional To Instrument Specific Controls

The signal-oriented, UUT-level Functional Instrument Control is mapped to the Specific Instrument Control level via logical resource translation, switch settings and Ada procedure calls.

The user test program calls made at the Functional Instrument Control level are concerned only with the logical resources present in the test system instruments themselves. These logical resources represent capabilities resident in the test station but do not call out the actual instruments themselves. The UATL gives these resources recognizable names as part of a logical device list:

```

type LOGICAL_DEVICE is (DIGITAL_TO_ANALOG_1,
                        DIGITAL_TO_ANALOG_2,
                        ELECTRONIC_COUNTER_1,
                        ELECTRONIC_COUNTER_2,
                        FREQUENCY_SYNTHESIZER_1,
                        FREQUENCY_SYNTHESIZER_2,
                        FREQUENCY_CONVERTER_1,
                        FREQUENCY_CONVERTER_2,
                        I1553_TESTER_1,
                        I1553_TESTER_2,
                        MULTIMETER_1,
                        MULTIMETER_2,
                        POWER_METER_1,
                        POWER_METER_2,
                        SPECTRUM_ANALYZER_1,
                        SPECTRUM_ANALYZER_2,
                        SWITCH_1,
                        SWITCH_2,
                        UATL_COMPUTER_1,
                        UATL_COMPUTER_2,
                        NONE);

```

The actual instruments supported by a particular test station must also be known to the system. The UATL gives them names as part of the physical device list:

```

type PHYSICAL_DEVICE is (BOONTON_4200
                        CDS_53A_352,

```

```

CDS_53A_453,
DEC_UVAX_11,
DEC_UVAX_3200,
DEC_VAXSTATION,
GIGATRONICS_1018,
GIGATRONICS_900,
HP_3438A,
HP_438A,
HP_5345A,
HP_5355A,
HP_59307A,
HP_59501A,
HP_8588A,
NONE);

```

In addition to the actual physical device to be used in the Functional Instrument Control call, the test station needs to know the address of that physical device on the 488 bus. In the UATL, the IEEE-488 interface software is programmable to handle instruments which terminate with CR,LF or instruments which use the service request as part of their operations. It also implements a protocol for communicating between computers hosting UATL based test programs.

The mapping between the logical and the physical resource is performed by reading a test station configuration file at program initialization. This is performed in the "body" of the Functional Instrument Control package. The following is an example of a UATL configuration file for mapping the logical resources to physical instruments with a given IEEE 488 address and protocol:

POWER_METER_1	HP_438A	13 SOL LF
FREQUENCY_SYNTHESIZER_1	GIGATRONICS_1018	06 SOL LF
MULTIMETER_1	HP_3438A	23 SOL LF
DIGITAL_TO_ANALOG_1	HP_59501A	02 SOL LF
SPECTRUM_ANALYZER_1	HP_8588A	18 SOL LF
SWITCH_1	CDS_53A_352	24 SOL LF

The first field is the logical resource, the second is the actual instrument used, the third is the IEEE-488 bus address, the fourth is the protocol used (SOL is solicited, which means the unit must be addressed to talk in order to get a response, it does not issue a service request), and the fifth is the message termination (linefeed for all these instruments). When this file is read the Functional Instrument Control also initializes the IEEE-488 interface software to operate according to the characteristics specified for the instruments in the test system.

The switch settings required to connect the physical device to the specified UUT connection are determined from another configuration file which contains information about the test station switching network. The required devices are switched in to accomplish the Functional Instrument Control call.

Each operation returns a completion status. This status indicates the success or failure of the requested operation. The UATL currently returns the following values for its operations:

```

type DRIVER_STATUS is (SUCCEEDED,
                        FAILED,
                        FUNCTION_NOT_SUPPORTED,
                        PARAMETER_OUT_OF_RANGE,
                        INVALID_MEASUREMENT,
                        NO_INSTRUMENT_RESPONSE,
                        INSTRUMENT_NOT_ON_BUS,
                        MEASUREMENT_UNDER_RANGE,
                        MEASUREMENT_OVER_RANGE,

```

WRONG_DEVICE_MODE,
 DEVICE_TIMEOUT,
 EXTERNAL_SIGNAL_NOT_RECEIVED,
 COMMAND_SYNTAX_ERROR,
 AMBIGUOUS_TRANSLATION,
 RESOURCE_NOT_GRANTED);

The final step is make the procedure call to the specific instrument with the appropriate bus address. The best way to describe this is by way of example. Consider the previous examples of the logical resource handling and switching as pertinent to this example and a call is made to apply an AC signal:

```

APPLY_AC_FREQUENCY
(USING    => FREQUENCY_SYNTHESIZER_1.
 FREQUENCY => 9.0E9.
 POWER    => -65.0.
 CONNECT_VIA => "YIG_IN".
 STATUS   => SYSTEM_STATUS);
  
```

This call wants to place a 9 Gigahertz, -65 dBm signal from FREQUENCY_SYNTHESIZER_1 at the "YIG_IN" pin of the UUT. The resource translation indicates that physical device GIGATRONICS_1018 at bus address 6 will be used. The switching logic connects the physical device to the UUT pin "YIG_IN". Then a call is made to the Gigatronics 1018 Frequency Synthesizer package procedure to GENERATE_FIXED_FREQUENCY:

```

GIGATRONICS_1018_FREQUENCY_SYNTHESIZER
GENERATE_FIXED_FREQUENCY
(BUS_ADDR    => BUS_ADDRESS.
 FREQUENCY   => FREQUENCY.
 OUTPUT_LEVEL => POWER.
 STATUS      => STAT);
  
```

where BUS_ADDRESS is equal to 6, FREQUENCY is equal to 9.0E9, and POWER is equal to -65.0. STAT will be the status returned to the caller.

By design the instrument level package procedure call looks very similar to the functional level call. This allows easy integration into the Functional Instrument Control scheme. This is adhered to as much as is practical, but when necessary deviations occur the functional instrument package will perform the translation from the standard functional level call to the unusual instrument requirements. Where possible, similar functions in different instrument level packages will also have the same procedure profile. This standardizes the Ada instrument driver interfaces which makes instrument substitution in the UATL test set very easy.

A further strength of this approach is that errors are trapped by the Ada exception mechanism at the level of the specific instrument package procedure calls. A call cannot be made to the instrument package with parameters that are out of range.

TEST MANAGER

The UATL Test Manager provides a menu driven environment in which a user can create and run UATL test and reduction/analysis programs. As illustrated in Figure 15, it organizes and controls the test station environment and provides a portable, menu driven, access to all the system facilities.

The test manager prompts the user to configure the system, run any test or test analysis program in the directories,

examine the test results, build a test or reduction/analysis program, create reusable test packages, or run any of the supplied utility programs. The test manager provides access to the UATL builder tool that guides a user through the steps to generate UATL test and data reduction programs, and to the trajectory editor that enables the user to create simulation test trajectories. The test manager menus also contain choices to edit, compile, link, delete, enter, and extract files.

The user does not even have to be concerned with the location of files or their full names. The UATL_Test_Manager_Translations package provides a set of functions that will translate a short (user supplied) name into a full file name that contains the directory and file extension.

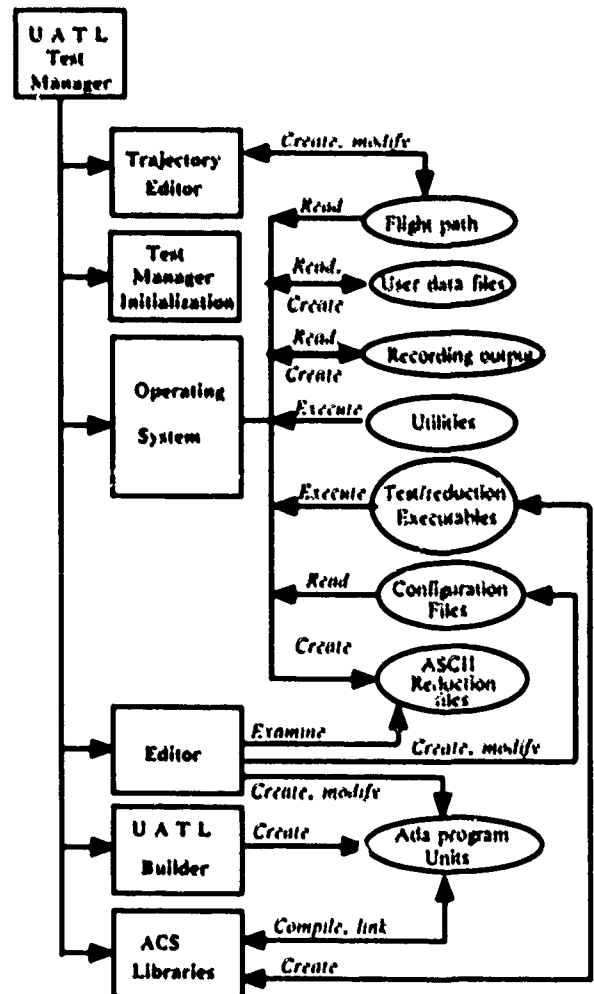


Fig. 15. Interface block diagram for the UATL Test Manager.

GENERATING A TEST PROGRAM

An applications programmer can generate a test program by writing an Ada main program procedure that uses the UATL components to initialize and control the desired test operation. To simplify the process for the non-sophisticated

user, a test program generation tool has also been developed.

A UATL based test program consists of a user generated "main" test procedure which controls the desired test functions and sequencing, and an optional modifier/comparator package which contains the tasks invoked for modifying stimulus or comparing response messages. The UATL test generation tool (Fig. 16) prompts the user for the desired test functions and automatically generates the "main" test procedure and modifier/comparator tasks package from the user supplied messages and modifier/comparator procedures.

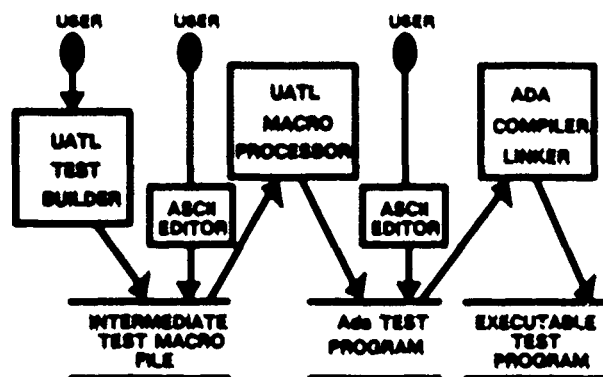


Fig. 16. Test Program Generation Sequence Using Automated Tool. User can input/modify the test program at each of the indicated points.

A macro language has been created as a part of the test generation tool to aid in writing test programs which require complex data structures, e.g. a list of stimulus messages. These macros simplify the specification of the stimulus, response, and response read messages, modifier/comparator tasks, trigger actions, and modifier/comparator task definitions.

The test generation tool consists of the Test Builder and the Macro Preprocessor. The builder is an interactive program which guides the user through writing a test program. The test procedure builder presents valid UATL functions and options for the user to select and supply the required input. The user inputs are checked for proper type and value before being accepted. The tool automatically "with"s in the appropriate UATL packages for the selected functions variables of the proper type for the user-supplied names. The builder then prompts the user for the information to define the modifier/comparator tasks. This information includes the user-defined procedures that will be called for the modify/compare operation, and the data type that it will process. The output from the test builder is a UATL test macro file consisting of Ada code with embedded macros. The macro processor inputs the macro file and outputs compilable, correct Ada code.

Fig. 16 also illustrates the test program generation and modification sequence. The user can enter the test writing process at any of the indicated points. The test program can be created using the interactive Test Builder tool, and modified or created by editing the intermediate simplified macro or Ada source files.

The generated Ada test program source files are then compiled in the following order: user-supplied messages, modifier/comparator procedures, modifier/comparator tasks, and test procedure. The linked object files result in an executable UATL based test program.

Trajectory Generator

This tool provides a menu driven editor for creating and modifying flight profiles for the UUT and other simulated units in the test environment. The flight profiles are defined by waypoints to simplify trajectory planning. Waypoints are defined by three dimensional position (x,y,z), speed, and turn acceleration. Two stage maneuvers are then computed between waypoints: constant rate and altitude turns heading towards the next waypoint (with coordinated roll and yaw), and constant linear acceleration to reach the defined speed at the next waypoint's position. This "Closed form" (non-integrated) kinematic solution is used to assure repeatability of trajectory scenarios. Use of a linked list waypoint data structure also supports dynamic flight profile changes during test operation.

Procedures are also provided to supply computed trajectory position, altitude, and velocity for the UUT or any of the other simulated units at requested test scenario time marks based on the defined test trajectories. The test programs can use this data to control the generation of test stimuli in real time with propagation delays and angular dispersions to simulate motion in the laboratory. This is very important if meaningful testing for functions that depend on UUT and interacting unit motion are to be verified in the laboratory.

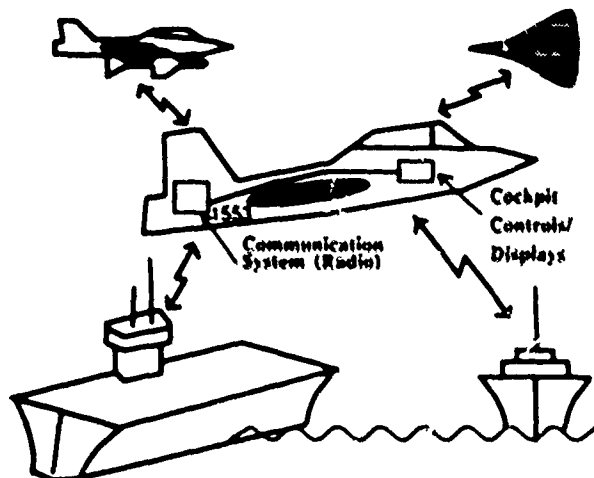
The trajectory computations package can also be used in post-test reduction analysis to verify the navigation accuracy of the recorded test data.

USING THE UATL

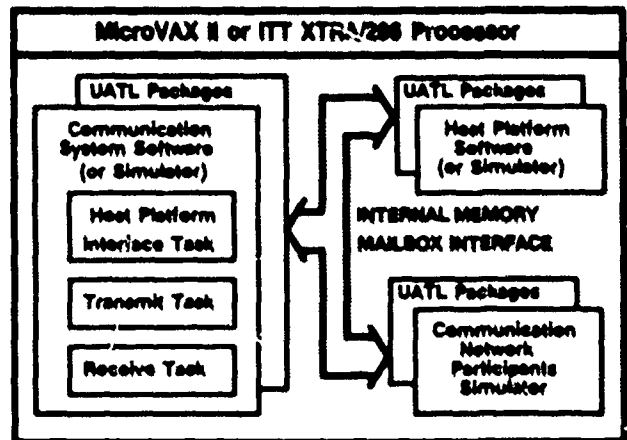
To verify the UATL software and to illustrate its capabilities, several software/system integration and fault detection/isolation demonstration tests were developed.

The software/system integration demonstrations (Fig. 17) included running a set of UATL test driver programs simulating a communication system interacting with its host platform and other network participants. In a typical system the host platform sends messages to the communication system over the 1553B bus for transmission over the RF network to other participants, and messages received from other participants by the communication system are sent to the host platform over the 1553B bus for processing and display to the operator. The flexibility in changing the interface over which the UATL and UUT can interact was used to easily change the test configuration as needed to various stages of the development process. For demonstration purposes the UATL was used to simulate each of the interfacing units in the communication system environment. At any point in the process the real software/system under test can be introduced and tested with the UATL simulating its environment.

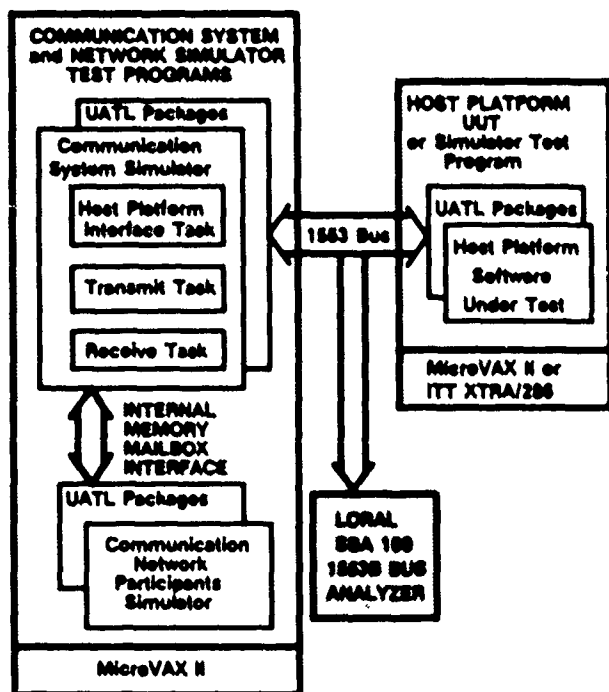
First software integration is performed for all system elements running in one processor. Then by placing the communication system and network participants simulators



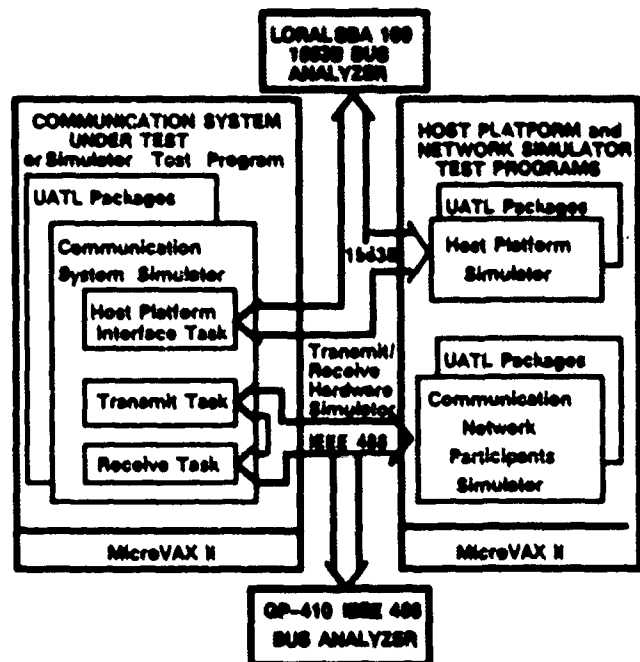
A. Typical Software/System Integration Problem



B. Integration within a Standalone Software Development Host Processor



C. A Host Platform Integration Test Configuration



D. Communication System Integration Test Configuration

Fig. 17. Demonstration of software/system integration testing of an avionics communication system. The UATL can be used to simulate the communication network, the host platform, and the communication system to drive the software/system under test in various configurations by using the appropriate internal memory or external 1553B and IEEE 488 interfaces.

in a processor containing a 1553B interface, it can be used for the integration testing of an actual host platform UUT. By placing the host platform and network participants simulators in one processor with a 1553B interface and an IEEE 488 interface emulating the RF hardware (or controlling a real RF transmitter/receiver), it can be used for the integration testing of an actual communication system UUT. For the demonstrations, a UATL based test program was used to also simulate the UUT in all the test configurations.

The factory/maintenance test demonstrations consisted of a test program running within a microVAX or ITT XTRA processor controlling a set of RF test instruments to characterize a breadboard containing two RF amplifiers or a breadboard YIG bandpass filter, and controlling a 1553B tester to perform a standardized test of the 1553B hardware installed in the MicroVax.

Ada IMPLEMENTATION CONSIDERATIONS

Some of the issues relating to the implementation of the UATL in Ada and the related design decisions are described below. These are related to the processing of test data by general, reusable, test packages and application specific test requirements with the strong Ada data typing and with portability issues.

Passing Generic Data (The Byte Array)

One of the critical items that had to be considered in designing the UATL was in providing a capability for performing the same stimulus/response functions for a variety of application dependent data types simultaneously over several external interfaces. If the UATL had been restricted to just concurrent, independent events the UATL could have been adequately implemented using generics to instantiate events for the data type required. Since these activities were to be concurrent and mixed, a common method of control was needed.

Thus a general byte array data format was defined to pass within the UATL stimulus/response functions.

```
subtype DATA_BYTE is NATURAL range 0..255;
type BYTE_ARRAY is array (NATURAL range <>)
  of DATA_BYTE;
pragma PACK (BYTE_ARRAY);
type BYTE_ARRAY_PTR is access BYTE_ARRAY;
```

The byte array is an unconstrained array of data bytes. User defined data objects are converted into the byte array using unchecked conversion. The general byte array form is treated merely as the memory image of the user-defined data type. No operations are performed on the byte array by the UATL other than to pass it to the appropriate destination at the appropriate time. The UATL only uses the length attribute of the byte arrays in its processing. This is analogous to doing low level I/O wherein the object's address and byte count is passed to some device to perform the I/O operation. When the byte array is passed to an interface for transmission it is converted to a general form for that particular interface. Messages received from a particular interface are converted from that interface's general form into the UATL general form of the byte array.

```
-- General message form for the 1553 interface
type I1553_MESSAGE
  (WORD_COUNT : DATA_WORD_COUNT_TYPE := 0) is
  record
    RT_ADDRESS   : ID_TYPE;
    COMMAND      : COMMAND_TYPE;
    SUBADDRESS    : SUB_ADDRESS;
    case WORD_COUNT is
      when 0 =>
        FULL_MESSAGE : WORD_ARRAY (1..32);
      when others =>
        MESSAGE       : WORD_ARRAY (1..WORD_COUNT);
    end case;
  end record;
```

The user programs process the application dependent data and the UATL processes the general reusable message control headers. Whenever this data appears at the user test level it is converted back into the user's data type. The UATL provides a set of generics to perform these

conversions in a consistent manner. The user instantiates the generic functions for the types to be converted and the UATL conversion generics take care of ensuring that it is done correctly.

```
-- Generic conversion function to convert any type to
-- an equivalently sized byte array type.
```

```
generic
  type UATL_SOURCE is private;
  function CONVERT_TO_BYTE_ARRAY
    (SOURCE : in UATL_SOURCE) return BYTE_ARRAY;
```

```
-- Generic conversion function to convert a byte array
-- to an undiscriminated type of equal or greater size
```

```
generic
  type UATL_TARGET is private;
  function CONVERT_FROM_BYTE_ARRAY
    (IN_ARRAY : in BYTE_ARRAY) return UATL_TARGET;
```

```
-- Generic conversion function to convert a byte array
-- to a type with one discriminant of equal or greater
-- size
```

```
generic
  type UATL_DISCRIMINANT_TYPE_1 is (<>);
  type UATL_TARGET (DISC1 : UATL_DISCRIMINANT_TYPE_1)
    is private;
```

```
function CONVERT_DISCRIMINATED_1
  (IN_ARRAY : in BYTE_ARRAY;
   DISC_1 : in UATL_DISCRIMINANT_TYPE_1)
  return UATL_TARGET;
```

```
-- Generic conversion function to convert a byte array
-- to a type with two discriminants of equal or
-- greater size
```

```
generic
  type UATL_DISCRIMINANT_TYPE_1 is (<>);
  type UATL_DISCRIMINANT_TYPE_2 is (<>);
  type UATL_TARGET
    (DISC1 : UATL_DISCRIMINANT_TYPE_1;
     DISC2 : UATL_DISCRIMINANT_TYPE_2) is private;
```

```
function CONVERT_DISCRIMINATED_2
  (IN_ARRAY : in BYTE_ARRAY;
   DISC_1 : in UATL_DISCRIMINANT_TYPE_1;
   DISC_2 : in UATL_DISCRIMINANT_TYPE_2)
  return UATL_TARGET;
```

The use of these conversions are limited to passing data at procedure and task calls. Even though unchecked conversions are used internally, if the UATL supplied conversion function for the correct number of discriminants in the user data is used (compiler checked) then the correct object is obtained. Additionally, the UATL builder tools are constructed to make the use of these conversions very easy.

Passing User Defined Procedures (Task Access Objects)

Another problem that had to be solved in developing the UATL was the need to pass user supplied procedures to perform application dependent stimulus modification, response and trigger comparisons, and data reduction analysis to the general reusable UATL test functions. Ada does not allow the passing of procedures as run time parameters. The brute force method would be to have fixed named procedure specifications that are called from the UATL for which the user provides application dependent bodies. This technique can run into large problems in

managing all the disparate Ada libraries with like-named user defined bodies.

A second option is to pass task access objects in place of the procedures. They give all the procedural functionality required and can be passed as parameters. The problem in the UATL domain becomes the need to pass dissimilar task access objects to a general set of stimulus/response functions. The UATL requirement for concurrent, mixed events precludes the use of generics in the UATL events.

It was thus decided to use a task type template concept to accomplish this requirement. The UATL defines a general form of the modifier and comparator tasks that contain two entry points; one for passing an access pointer to the byte array message being passed to/from the task, and another to terminate the task gracefully. The message parameter is defined as "in out" for the modifier task to allow its modification, and as "in" for the comparison task with an "out" parameter for the result of the comparison match operation. The template task body has the two entry points in a select statement encased in a loop. Included in this definition is an access type to the general task type template.

The user writes the desired modifier/comparator tasks in the form of the UATL task type template and includes the application dependent processing in the task body. The input byte array message is converted to his own object type using the UATL conversion generics. For the modify operation the user modifies the message in his own type definition, and then converts it into the byte array for passing to the UATL stimulus task. For the compare operation the user checks or extracts whatever fields he chooses from the message in his own type definition, and sets the returned match condition parameter to the UATL response/trigger task. The modifier/comparator tasks can communicate values among themselves by sharing a common data area. A comparator task can save a value from a UUT message field that a modifier task can then use to modify an outgoing message. This provides the real time "closed loop" test data operation capability.

The last step in the process is to pass the user's tasks to the UATL event tasks so that they can be called when required. This is accomplished by an unchecked conversion of the user's modifier/comparator task access object into the task access object which the UATL events are expecting. Since the entry point specifications and parameter profiles are identical, and the task body operation similar, the UATL events communicate with and control the operation of the user's task as if it were a task of the UATL general form.

-- UATL Modifier/Comparator Task Templates and Types

-- General modifier task type template.

```
task type UATL_MODIFIER_TASK is
  pragma PRIORITY (PRIORITY'last);
  entry MODIFY (GUM : in out BYTE_ARRAY_PTR);
  entry TASK_COMPLETE;
end UATL_MODIFIER_TASK;
```

-- General modifier task body template.

```
task body UATL_MODIFIER_TASK is
begin
  loop
    select
      accept MODIFY (GUM: in out BYTE_ARRAY_PTR) do
        null;
```

```
      end MODIFY;
    or
      accept TASK_COMPLETE;
        exit;
      end select;
    end loop;
  end UATL_MODIFIER_TASK;

-- Access pointer to general modifier task
type UATL_MODIFIER_TASK_ACCESS is
  access UATL_MODIFIER_TASK;

-- List of access pointers to general modifier task
type UATL_MODIFIER_TASK_LIST is array
  (POSITIVE range <>) of UATL_MODIFIER_TASK_ACCESS;

-- Access pointer to list of access pointers to general
  modifier task
type UATL_MODIFIER_TASK_LIST_ACCESS is
  access UATL_MODIFIER_TASK_LIST;

-- General comparator task type template.
task type UATL_COMPARATOR_TASK is
  pragma PRIORITY (PRIORITY'last);
  entry COMPARE
    (GUM : in BYTE_ARRAY_PTR;
     MATCH_STAT : out UATL_MATCH_STATUS);
  entry TASK_COMPLETE;
end UATL_COMPARATOR_TASK;

-- General comparator task body template.
task body UATL_COMPARATOR_TASK is
begin
  loop
    select
      accept COMPARE
        (GUM : in BYTE_ARRAY_PTR;
         MATCH_STAT : out UATL_MATCH_STATUS) do
        MATCH_STAT := MATCH;
      end COMPARE;
    or
      accept TASK_COMPLETE;
        exit;
      end select;
    end loop;
  end UATL_COMPARATOR_TASK;
```

For the actual generation of a test program, the UATL Builder tool relieves the user of having to define these template tasks. The tool is structured so that the user supplies a package of modify or compare procedures that he wants to use. The specification of the modify procedures has one "in out" parameter of the user's message type. The compare procedure has an "in" parameter of the user's message type and an "out" parameter of the resulting match status. The UATL builder tool will automatically construct a package of task types and task access types with the necessary conversions to call the user procedure with the message that it is expecting.

The interactive builder tool prompts the user for the necessary information to construct these tasks and creates a user-editable macro file as shown below.

-- User editable Modifier/Comparator macro examples

--- Modifier Task

```
@MODIFIER_NAME    => MOVE_POSITION
@CALL_PROCEDURE   => USER_MOVE_PROCEDURE
@PASS_MESSAGE_TYPE => I1553_MESSAGE
@DISCRIMINANT_TYPE => DATA_WORD_COUNT , 0
```

--- Comparator Task


```

@COMPARATOR_NAME => CHECK_POSITION
@CALL_PROCEDURE  => USER_CHECK_PROCEDURE
@PASS_MESSAGE_TYPE => I1553_MESSAGE
@DISCRIMINANT_TYPE => DATA_WORD_COUNT . 0

```

As shown below, the UATL Macro Processor then automatically expands this macro file into compilable Ada code containing the correct modifier/comparator tasks. The macros are included in the Ada code as comments.

```

-- Modifier Task specification
--- Modifier Task
-----
-- @MODIFIER_NAME      => MOVE_POSITION
-- @CALL_PROCEDURE     => USER_MOVE_PROCEDURE
-- @PASS_MESSAGE_TYPE  => I1553_MESSAGE
-- @DISCRIMINANT_TYPE  => DATA_WORD_COUNT . 0
-----

task type MOVE_POSITION_TYPE is
  pragma PRIORITY(PRIORITY'last);
  entry MODIFY (CUM : in out BYTE_ARRAY_PTR);
  entry TASK_COMPLETION;
end MOVE_POSITION_TYPE;
type MOVE_POSITION_TYPE_ACCESS
  is access MOVE_POSITION_TYPE;
MOVE_POSITION : MOVE_POSITION_TYPE_ACCESS;
function CONVERT is new
  UNCHECKED_CONVERSION (MOVE_POSITION_TYPE_ACCESS,
    UATL_MODIFIER_TASK_ACCESS);

-- Modifier Task body
task body MOVE_POSITION_TYPE is
  MESSAGE : I1553_MESSAGE;
  function CONVERT is new CONVERT_TO_BYTE_ARRAY
    ( I1553_MESSAGE );
  function CONVERT is new CONVERT_DISCRIMINATED_1
    ( DATA_WORD_COUNT , I1553_MESSAGE );
begin
  loop
    select
      accept MODIFY (CUM: in out BYTE_ARRAY_PTR) do
        MESSAGE := CONVERT( CUM.all, 0 );
        USER_MOVE_PROCEDURE ( MESSAGE );
        CUM.all := CONVERT( MESSAGE );
      end MODIFY;
    or
      accept TASK_COMPLETION;
      exit;
    end select;
  end loop;
end MOVE_POSITION_TYPE;

-- Comparator Task specification
--- Comparator Task
-----
-- @COMPARATOR_NAME    => CHECK_POSITION
-- @CALL_PROCEDURE      => USER_CHECK_PROCEDURE
-- @PASS_MESSAGE_TYPE   => I1553_MESSAGE
-- @DISCRIMINANT_TYPE   => DATA_WORD_COUNT . 0
-----

task type CHECK_POSITION_TYPE is
  pragma PRIORITY(PRIORITY'last);
  entry COMPARE (CUM : in BYTE_ARRAY_PTR;
    MATCH_STAT: out UATL_MATCH_STATUS);
  entry TASK_COMPLETION;
end CHECK_POSITION_TYPE;
type CHECK_POSITION_TYPE_ACCESS
  is access CHECK_POSITION_TYPE;
CHECK_POSITION : CHECK_POSITION_TYPE_ACCESS;
function CONVERT is new
  UNCHECKED_CONVERSION(CHECK_POSITION_TYPE_ACCESS,
    UATL_COMPARATOR_TASK_ACCESS);

-- Comparator Task body
task body CHECK_POSITION_TYPE is

```

```

function CONVERT is new CONVERT_DISCRIMINATED_1
  ( DATA_WORD_COUNT , I1553_MESSAGE );
begin
  loop
    select
      accept COMPARE
        (CUM : in BYTE_ARRAY_PTR,
         MATCH_STAT : out UATL_MATCH_STATUS) do
        USER_CHECK_PROCEDURE
          (CONVERT( CUM.all, 0 ), MATCH_STAT);
        end COMPARE;
    or
      accept TASK_COMPLETION;
      exit;
    end select;
  end loop;
end CHECK_POSITION_TYPE;

```

The user then declares the UATL events with these constructed tasks as the modifier/comparator tasks to be called. Thus the user never has to worry about the general UATL modifier/comparator task form but only about the logic inside his own modifier/comparator procedures.

```

-- UATL stimulus call using the defined modifier
STIMULUS (BLOCK      => "STIMULUS CALL BLOCK",
  EVENT      => "STIMULUS CALL EVENT",
  INTF      => I1553B,
  STIM_LIST  => USERS_MESSAGE,
  CIRCULATE  => 3,
  MSG_RATE  => 2.0,
  TIME_TO_GO => 3.0,
  MODIFIER   => CONVERT(MOVE_POSITION),
  CYCLE      => CYCLE_CONSTANTLY,
  TRIG_MODE  => RUN);

```

Functional Instrument Control Data Typing

In defining the instrument control data types a tradeoff had to be made between strong typing that would allow tight control of the legal operations on the defined types and would catch more possible errors at compile time, and the amount of effort needed to implement this. While it is desirable to define all physical quantities (e.g. volts, amperes, watts, meters, etc.) as separate types and explicitly define all the allowed operations on these types, it becomes unwieldy to achieve this in practice.

While the legal operations for data types can be defined, if tight control is to be achieved it is necessary to define all the possible operations on them. This can be a daunting task if one considers all the intermediate results possible in a very complex calculation. An answer to that might be to allow a general type like float to be the type of those intermediate results, but then that significantly weakens the typing for the main types defined.

Additionally, although operations between operands of the same type are in general not valid for physical quantities (e.g. volts * volts is undefined), they are implicitly allowed in Ada. Infix operator functions that would hide the implicit operations would have to be written to signal some error (an exception, perhaps) as part of their execution for invalid operations between operands of the same type.

Specific infix operators would also have to be defined for all physical types to allow their multiplication/division with unit-less float types. A further problem is then encountered in the use of named numbers since the context resolution is

ambiguous when using numeric literals with overloaded operators. The literals would have to be explicitly converted to unit-less float to avoid ambiguity with conversion to the invalid, error producing, operation between operands of the same type. The ramifications of strong typing seemed to outweigh the benefits obtained.

The UATL solution was to define the data types for all physical units as subtypes of the portable float type. Although this does not prevent the use of physically invalid mathematical operations between types, it still separates the definitions of all physical units and defines them with the proper ranges that are verified by the Ada constraint checking. The use of descriptive names for variables, with the physical subtype name at the end, e.g. `POWER_LEVEL_DBN` also helps minimize data typing errors.

Rehosting Issues

The UATL was developed on a VAXstation II and then ported to the ITT XTRA/286. In general, Ada's portability enabled us to rehost the UATL with a minimum of host processor unique code.

Problems were encountered with the MS-DOS 640K memory management limitations. Part of the problem was solved by using the Alsys compiler which was able to create executables that can run in protected/extended memory mode. However this could not be done when interfacing with the IEEE 488 bus because the interface driver supplied by National Instruments only operated in real mode.

The UATL uses unchecked conversions but not all compilers perform the conversion in the same manner. Problems were encountered with the unchecked conversion of variant records in the Meridian 2.1 compiler. Because the Meridian compiler stores variant records as a set of fixed fields with a pointer to the variant portion. When an unchecked conversion is performed the user gets the address of the variant field instead of the actual data within the field. Special conversion packages were written to extract the desired data. The results of an unchecked conversion on the Alsys and DEC compilers do provide the same expected data and so the same generic conversion package can be used.

ACKNOWLEDGMENTS

This work was funded by a STARS (Software Technology for Adaptable Reliable Systems) Foundations Contract No. N60921-87-C-0285 under the management of USAF Col. Joseph Greene, Director of the STARS Joint Program Office, Ms. Elizabeth E. Wald, STARS Navy Deputy Director for Ada Foundations Technology at the Naval Research Laboratory, and Mr. Phillip Q. Hwang, our project COTR and Director of the ACSAD Computer Research Laboratory at the Naval Surface Warfare Center.

REFERENCES

- [1] DoD Directive 3405.2, *Use of Ada in Weapons Systems*, March 30, 1987.
- [2] DoD Directive 3405.1, *Computer Programming Language Policy*, April 2, 1987.
- [3] Roy T. Oishi, "ATLAS EXTEND, Its Effect on ATE System Software", in *AUTOTESTCON '88 Symposium Proceedings*, IEEE Press, 1988 (88CH2575-9).
- [4] ANSI/IEEE Standard ATLAS Test Language 416-1984 (S1109696), and ANSI/IEEE C/ATLAS Test Language 716-1985 (S1110231), IEEE Computer Society Press.
- [5] A. Bunsen, Investigation of Programming Languages for Automatic Testing Applications, in *Proceedings of AUTOTESTCON '87*, IEEE Press, 1987 (87CH2510-6).
- [6] J. Foreman and J. Goodenough, *Ada Adoption Handbook: A Program Manager's Guide*, Technical Report SEI-87-TR-9, Software Engineering Institute, May, 1987, pg. 45.

BIOGRAPHIES

All the authors are members of the Software Engineering staff at ITT Avionics, ITT Defense Technology Corporation, 390 Washington Avenue, Nutley, NJ 07110.

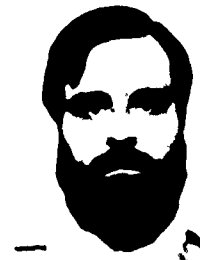


Leonard D. Mollod is the Director of Electronic Defense Software Engineering at ITT Avionics. He is responsible for the software in ITT's electronic countermeasures and electro-optics product lines, and also directs the Advanced Ada Technologies and sensor fusion IR&D efforts. Prior responsibilities have included the management of the ITT's integrated communications - navigation - and identification projects including ICNIA and JTIDS. Mr. Mollod has been awarded three U.S. patents and received the M.E.E. and B.E.E. from the City College of the City University of New York.



Jehuda Ziegler is a Senior Technical Consultant responsible for the Advanced Ada Software Technologies group. This includes the management of the STARS Ada Foundations project to develop the Universal Ada Test Language, the prototype ASPJ WRA-3 UATL/Ada production tester project, and several Ada IR&D projects. Dr. Ziegler received a PhD and MS in Physics from New York University in 1975 and 1971, a BS, Magna Cum Laude with Honors in Physics, from Brooklyn College in 1969, and was elected to Phi Beta Kappa and Alpha-Sigma Lambda Honor Societies.

Previously he was the Software Project Leader for the ICNIA ADM and JTIDS DTDMA FSD projects, and has successfully led them through all phases of the software development process; from requirements definition through design, code, integration, and qualification testing. He has proposed and defined software design/development tools and methods to increase the productivity and quality of the software development process.



Jerry M. Grasso is a Principal Member of the Technical Staff responsible for the technical leadership of the STARS Foundation Area project to develop a Universal Ada Test Language (UATL). Mr. Grasso received a BA in Mathematics from the State University of New York, Stony Brook in 1978 and has several years of experience in Ada software development.

He implemented the CNI Rapid Turnaround Support System for analyzing the effectiveness of threats against several CNI systems, and several interface routines for hosting an Ada Distributed Multiprocessor Executive on Motorola 68010's. On the ICNIA project, he completed the development, rapid prototyping, and integration of the JTIDS TDMA software in Ada.

Previously he successfully lead the integration team in the completion of the software qualification tests for the Navy JTIDS FSD project, and was also responsible for the Executive, PPLI data link control, and relative navigation functions.



Linda G. Burgermeister is a Member of the Technical Staff responsible for developing much of the software on the UATL project. Ms. Burgermeister received an MS in Computer Science in 1988, and a BE in Chemical Engineering in 1983 from the Stevens Institute of Technology and has several years of experience in Ada software development.

Ms. Burgermeister implemented the UATL test manager, data recording, ASCII and Graphical data reduction, and internal mailbox communication functions. She is also responsible for the UATL configuration management and has helped develop the UATL documentation. Previously, Ms. Burgermeister has developed real-time embedded software for ASPJ (Airborne Self-Protection Jammer), and the VHF/UHF communication software in Ada for ICNIA.

A DIANA Query Language for the Analysis of Ada Software

Christopher Byrnes

The MITRE Corporation

ABSTRACT

The Descriptive Intermediate Attributed Notation for Ada (DIANA) Query Language (DQL) is a set of primitive search operations and combining operators for querying the DIANA intermediate form of Ada source code, much like a conventional information system data base can be queried. DQL can be used by an analyst to determine how well the Ada software conforms to design standards, to compute metrics based on the software's structure, and to browse through the software. DQL can also be used as an integration layer on which new and existing tools may be implemented to extract relevant information about Ada source code. DQL is being implemented on a network of Sun 3[®] workstations as an Electronics Systems Division (ESD) Acquisition Support Environment (EASE) utility using a bit-mapped, windowed user interface and a server-based DIANA tree manager to allow interactive analysis of the Ada source.

1 INTRODUCTION TO THE DIANA QUERY LANGUAGE

1.1 DIANA

The Descriptive Intermediate Attributed Notation for Ada (DIANA) is a public standard (Evan83) for defining a tree structure that captures all the semantic information in Ada source code. Semantic tree structures such as DIANA are often produced by the front end of a compiler, allowing different compiler back ends to be customized for a particular target computer. In DIANA's case the trees become the mechanism by which the Ada compiler "browses" the source code during activities such as code generation. The creators of DIANA recognized that such an intermediate form would be useful to tools other than the compiler. The DIANA standard is an attempt to encourage compiler writers to generate information that could be used by tool developers.

1.2 THE DIANA QUERY LANGUAGE

The DIANA Query Language (DQL) and its associated tools are a mechanism to analyze Ada source code (or Ada used as a Program Design Language) by searching through the Ada source for language constructs of interest. Using an Ada intermediate

form such as DIANA eliminates the need for these analysis tools to parse and check for semantic correctness; tools that build DIANA trees perform this function. DQL and its tools allow the querying of DIANA trees based on Ada language constructs. DQL does not require familiarity with DIANA and its model of connected nodes and clauses; it merely requires familiarity with the Ada language and its constructs as defined in the Ada Language Reference Manual (DOD83).

2 PROBLEMS BEING ADDRESSED BY DQL

2.1 ANALYSIS OF SOFTWARE WRITTEN IN ADA

An important step in the analysis of Ada software systems is the checking of that software against criteria that define the requirements for the reliability, maintainability, and other aspects of the system. The Ada language standard (MIL-STD-1815A) and compilers validated against that standard provide automatic checking (such as strong type checking) not found in earlier languages. But, there is a need for analysis capabilities that are outside the Ada standard and the compiler.

Each Ada application program is subject to its own set of criteria that defines the standards for how well written the software is. A development organization's project management approach, the software design methodology, the particular application's needs, and the preferences of the developers all combine to define these criteria. Some criteria can be expressed as explicitly defined design and coding rules; others are heuristics based on previous experience. Checking any large collection of software against a variety of rules is difficult, especially given the limited amount of time generally allocated to quality analysis. Automated support for such checking is needed, and fortunately it is possible because Ada's notations allow more of the information about the software to be captured.

2.2 CHECKING CONFORMANCE TO DESIGN RULES

Many application areas and design methods have formally defined design rules that can be mechanically applied to software. Examples include the Normal Form rules for data bases (Date82), the starvation and deadlock guidelines for communicating sequential processes (Hoar85), and the equivalency rules for data flow decompositions (Ward85). Some of these design rules are very specific to Ada, such as exceptions not outliving their names or "erroneous" assumptions (Soft82). These design rules involve checking the Ada code against itself and against other development products, such as a requirements specification.

• Sun 3 is a trademark of Sun Microsystems, Inc.

The process of defining all the design rules (coming from several different areas) as they apply to Ada is a major activity. Applying all these design rules to the Ada source code is also a major activity. Because of the importance of these rules in determining overall software quality, a large software development organization may use an independent Software Quality Assurance (SQA) group to ensure these rules are applied before the software is released. Manual application of design rules to software can be tedious (since much of it is mechanical), so automated support would aid SQA in its work.

DQL allows design rules to be formally stated as a DQL query. These rules can represent generally good software engineering practices, or they can be specific to an application domain. These queries can then be applied to the software to find those Ada code sections that conform to or violate these design rules.

2.3 BROWSING THROUGH ADA SOFTWARE

In addition to analysis with respect to the formally defined design rules, Ada software is subject to heuristic analysis. This heuristic analysis is frequently applied by having someone read the Ada software, looking for certain characteristics or patterns. This can be difficult to do, since in a well-modularized Ada system the information needed may be spread throughout many source files, and in several places within a given source file. The analyst will want to "browse" various specifications and bodies, parent and subunits, callers and callees, type declarations and uses, etc. Trying to browse manually through a large Ada system can be difficult, time consuming, and error prone.

Browsing in Ada is further complicated by overloaded names, scope and visibility rules, and renaming declarations. Browsing requires looking at the semantic meaning of an identifier, not just its syntactic appearance in the source code. Without semantic-based browsing, accurately reading Ada source code can be tedious. DQL allows semantic browsing by allowing an analyst to follow the connections between Ada source constructs (by "walking" the associated DIANA tree).

2.4 PORTABILITY OF ANALYSIS ACROSS HOST SYSTEMS

Since the Ada language was designed to maximize portability across environments, the formal and heuristic design rules that can be applied to Ada must also be portable. There exist some formal rule checkers (such as set/usage analyzers) and textual browsers (such as regular expression searchers) that can be useful for Ada, but many of these are specific to a particular host environment or a specific Ada Programming Support Environment (APSE). Proper application of design rules may involve comparative analysis of Ada software residing on different APSEs (e.g., for reusability or interface checks). Analysts would like to do their work without having to learn the individual Ada source code browsing procedures of several different hosts or APSEs.

DQL provides an abstract interface to the DIANA intermediate forms that can be created by a variety of host environments and APSEs. The same DQL query and design rule checking can be used across different DIANA implementations. DQL and its tools also allow queries that span DIANA trees so the Ada constructs in two different APSEs and/or hosts can be checked.

2.5 ACCESS TO SOFTWARE OVER A NETWORK

The community of software engineers wishing to analyze a particular piece of Ada software may be large. In addition to SQA, individual programmers, maintainers, and managers will want to apply design rules and browse the code. Ada programmers make heavy use of an Ada compiler as a quick measure of code that was just written. When available, automated design rule checkers used by programmers, managers, and SQA will also become frequently used tools. This will introduce the usual problems of multiple access across a network, and maintaining consistency among software stored at several different locations. Programmers browsing through Ada code would like the transitions between files and networks to be as smooth as possible.

The DQL tool set separates the activity of searching (large) DIANA trees with a DQL query from the display of any results from that query. The large amount of work associated with managing DIANA trees is isolated in a few DIANA tree servers, which are accessed by users at workstations on a local area network. DIANA tree servers allow users to share query results without forcing each user to recompute each DQL query on their own workstation.

3 SIMILAR WORK ADDRESSING THE PROBLEM OF ADA ANALYSIS

3.1 EXISTING DIANA ACCESS METHODS

Most Ada compilers that use an intermediate form like DIANA (or something similar to it) have defined a method for accessing nodes in the tree. Unfortunately most compiler vendors treat these access methods as proprietary, preventing other programs, such as analysis tools, from taking advantage of the DIANA trees. A few vendors allow partial access methods, such as the Intermetrics Program Library Access Package (PLAP) (Gord83) and the Rational® Design Facility (Bach87), but these are oriented toward offline report generation and not online browsing. Some DIANA access is provided on Rational's R1000® environment, but there is no vendor-independent DIANA tool.

3.2 EXISTING ADA ANALYSIS TOOLS

Current APSEs may contain tools that provide some information useful to an analyst. Some documentation tools such as Byron® (Gord83) and the Ada-based Design And Documentation Language (ADADL) provide Government standard deliverables and reference reports that have some use, but these are batch oriented, and it is difficult to correlate multiple batch-generated reports together while checking the satisfaction of a design rule. Other tools allow the computation of complexity measures and other metrics (Park87), but often these metrics are hard-coded to a particular formula and thus difficult to customize to a specific application's needs.

© Rational and R1000 are trademarks of Rational, Inc.

© Byron is a trademark of Intermetrics, Inc.

3.3 EXISTING ADA DESIGN RULES

The Ada literature contains many examples of design rules. Ada-specific design methodologies such as Buhr's System Design with Ada (Buhr84) and Cherry's Process Abstraction Method for Embedded Large Applications (PAMELA) (Cher86), define formal and heuristic rules to be applied by the designer. Other work (Nuss84) defines rules that are new to Ada or are modifications to traditional rules. The underlying methodology necessary to support Ada analysis is in place; what is needed is the technology to do it.

4 SIMILAR TECHNOLOGY FROM OTHER FIELDS

4.1 RELATIONAL INFORMATION SYSTEMS DATA BASES

The problems of supporting *ad hoc* data base searches, new types of reports, and checks on data base correctness are not new to the information systems community. They have developed data base forms such as relational data bases (Date82), Entity Relationship Attribute (ERA) models (Chen76), and semantic data networks (Hamm81) to structure their data in a way to support a variety of data access methods and reports. With the types of reports supported purposely left open-ended, a data base analyst can create ways to search for a variety of information patterns that give new insight about some enterprise.

4.2 RELATIONAL ALGEBRA

Data base analysts are aided by a standard set of mathematical methods that define what the attributes of the data base are, and how those attributes can be queried and combined. For relational data bases, a relational algebra has been defined that describes how queries can be formed, combined, nested, and stored. From the set of low-level operations provided by the relational algebra, an analyst can build a query that returns some aspect of the data base.

4.3 STRUCTURED QUERY LANGUAGE (SQL)

For relational data bases, a standard method has been created to allow applications programs to query a data base using a relational algebra. The Structured Query Language (SQL) (Date82) is a standard that defines the data types and the operators that implement the algebra. If a data base analyst can create a query in the relational algebra that checks some rule, then an applications program that automatically performs the rule check can be written.

4.4 ABSTRACTING IMPLEMENTATION DETAILS AWAY FROM THE USER

One goal of a data base query language is to abstract away the implementation details of the underlying data base. Queries can be formed directly from the algebra (as in the natural language query forms), or from applications programs (such as those using SQL) without regard for how the data is stored on a disk or how it is distributed around a network. Ada analysts working with source code would similarly like to browse without regard for directories, file positions, and network paths.

4.5 SYNTHESIZER GENERATOR

The Synthesizer Generator (Kope84) provides a semantic data base that is integrated with a source code editing system for a variety of computer languages. The Synthesizer Generator does not separate the functions of creating the software from analyzing the software for semantic correctness; this makes it a useful tool during source code creation and editing. This integration of editing and analysis is very light, making it difficult to perform just the analysis functions without working through the editor as well.

5 DQL IMPLEMENTATION APPROACH

5.1 DIANA TREES AS DATA BASES

DQL allows Ada source code (through its underlying DIANA trees) to be treated like a data base by defining a query language that operates on that data base. DQL has an advantage over query languages for information systems data bases in that the data definition language for Ada (DIANA) is a fixed standard, so there is little need to support user-defined data base schemas. DQL could be extended to handle well-defined extensions to DIANA such as the structured Ada comments used by ANNA (Luck84) and Byron (Gord83).

5.2 INTERACTIVE QUERY AND RESPONSE

The DQL implementation under development at MITRE is designed to provide (almost) immediate response to an Ada analyst's query. On-line browsing of Ada source requires that the user not be frustrated by long delays before an answer; otherwise a train of thought may get lost. If the cost (in response time) for applying design rules is small, an Ada programmer will be encouraged to apply these rules frequently and as a result, problems will be caught closer to the time when they are created.

5.3 EXAMPLES OF QUERIES

A DQL query is built up from approximately 350 primitive queries (one for each major type of DIANA node and attribute representing Ada source). In the examples below, bold indicates a reserved DQL keyword or character and *Italic* indicates a placeholder for an identifier or a query. A primitive query is of the form

```
search primitive_operator target;  
-- DQL has Ada-style comments
```

where *search* names the entities to search for, *target* defines where to find entities, and *primitive_operator* defines the type of entity (DIANA node type) to search for. *Search* and *target* can be character strings (to match Ada identifiers), the keyword *all* (to match anything), source file ranges, named DIANA nodes, named results, or another subquery whose results become the operand. For example, the following query

```
all exceptions_raised_in <DIANA node>;  
-- node identifies specific place
```

would return the DIANA node names which uniquely identify the name of any exception which is raised in the given *DIANA node* or in any node that is enclosed within that node's scope. A nested query would look like

```
(search operator1 target1) operator2 target2;
```

where the query within the parenthesis is called a subquery. An example of this is

```
(all exceptions_defined_in <node 2>)
exceptions_raised_in <DIANA node 1>;
-- query spanning two lines
```

which would again return exceptions raised in the scope of node 1, but here only those exceptions that are defined in the scope of node 2 would be returned. The subquery acts as a qualifier to the search operand of the outer query. A *primitive operator* is one of the 350 reserved keyword queries such as *object_defined_in*, *access_type_of* and *entry_calls_to*.

In addition to the subquery construct, DQL queries can be combined by using unary and binary operators. The unary operators are applied with *unary_operator* (*query*). The seven unary operators include *uniq*, which removes duplicate nodes, and *only*, which limits the searching of a scope to only its topmost level. For example, the query

```
uniq (all subprogram_calls_in <DIANA node>);
```

would return the nodes that identify a call to any subprogram, but any duplicate results that called the same subprogram would be eliminated by *uniq*. The binary operators are applied with (*query binary_operator query*). The six binary operators include *union*, which joins the results of two queries, and *diff*, which removes results present on both the left and right sides. For example, the query

```
((all subprogram_address <source range>) union
(all entry_address <source range>));
```

will return the nodes that identify any place in the given *source range* where the ADDRESS attribute is referenced for either a subprogram or a task entry.

Both query results and cursors (symbolic names for a given DIANA node) can be named in the form

```
result := query; cursor := <DIANA node>;
-- result & cursor definition
```

with later queries able to use those names. Query names (that act as macros) are defined in the form

```
name is query; -- query name definition
```

where current positions are used to replace placeholders in *query*. Queries are strongly typed so the user is warned when semantic DQL errors are made. In addition to searching for nodes, the count unary operator and the four standard mathematical functions (binary operators +, -, *, /) can be used to compute metrics.

5.4 INTERMEDIATE RESULTS AS STREAMS

Each query (including each subquery within a higher level query) produces a stream of results. These streams can be built and combined dynamically. Any additional connections needed

when crossing Ada library and source directory boundaries are transparent to the user. At the top-most level the query returns a stream of DIANA nodes (including source positions), so other software development environment tools can scroll editors, highlight text, or just list results as needed.

5.5 SERVER ARCHITECTURE TO PROMOTE PORTABILITY

DIANA trees are handled within DQL servers, with a centralized data base keeping track of which trees are handled by which server. When a query causes DIANA tree boundaries to be crossed, this may result in host processor boundaries being crossed as well. These different host processors may be different computers, running different operating systems, and so have different versions of the DQL server. DQL supports portability by using a standard Remote Procedure Call (RPC) interface that allows queries (and result streams) to cross arbitrary host boundaries.

6 IMPLEMENTATION CONCERNS

6.1 SIZE OF DIANA TREES

Our DQL implementation uses the Intermetrics Ada Compilation System (ACS) front end (Inte86) as the source of semantically complete DIANA trees. Like most complete DIANA trees, the ACS's intermediate form has a very large expansion factor (up to 20:1) from the original Ada source. Such large trees are time consuming to create and too large to allow each user to have his/her own copy. The motivation for having DQL servers was to allow these trees to be created once and then shared among all users. Figure 1 below shows a Muhr notation example of a query that was initially handled by Tree Server #1 running in Server #1, but eventually involved other Tree Servers (#2 and #3), running on another server, in the computation of the results for the user at Workstation #1.

6.2 DQL QUERIES CAN BE VERBOSE

A DQL query that checks for a particular design rule can become very complex, involving many subqueries. Just as information systems users have trouble instantly creating semantically correct SQL queries, Ada analysts may have trouble creating complex DQL queries by just typing them in by hand. A menuing system that provides an analyst with some guidance is being developed. DQL also allows individual queries to be stored by name; this allows a complex query to be built up gradually from its components and to be reused easily.

6.3 BROWSING THROUGH INTERMEDIATE RESULTS

Once the query results have been computed, the analyst will wish to browse through them and the source code that they represent. The applications program that captures the highest level result stream and displays it on a workstation screen is integrated into EASE, an Ada analysis environment (Byrn88) running on Sun 3 workstations that allows multiple windows of information to be created and managed. The DQL programs become just another type of analysis tool supported by EASE.

Figure 1 shows a DQL Query Results task (program) that will display results (as ASCII text) as they are received from the server. The user can start using these results as soon as they are received; there is no need to wait for the query to be completed. The user could request an editor (starting at the line containing a particular node) or start a new query based on a particular result or an entire stream (using it as the source or target).

6.4 CONCURRENT ACCESS BY MULTIPLE USERS

Several different users may wish to analyze the same DIANA tree at the same time. Forcing some of them to wait until an earlier user has finished using the tree would introduce unacceptable response time delays. The DQL server implementation uses dynamically created Ada tasks to do each of the subqueries. Figure 2 below shows how an example query in a server might connect its subqueries. The concentrators combine individual subqueries into a single stream of results. Another user's query would introduce new tasks, but the server would allow all of them to run concurrently.

Note that the connectivity between result streams is done almost totally within a DIANA tree server. Most new queries will be refinements of previous results; dynamically added queries can reuse existing subqueries and results. Potentially expensive net-

work transfers of results to a workstation are done only when the user explicitly requests them with the unary display (query) function. The workstation can be dedicated to the user interface, while the server's hardware can be as powerful as necessary to act as a DIANA data base machine.

6.5 MANAGEMENT OF EARLIER QUERY RESULTS AND DEFINITIONS

As an analyst uses DQL, many query definitions, cursor names, and query results may be created. In addition, predefined query definitions and results (capturing an organization's standard design rules) may be used. To manage all this information, special EASE status windows (on the user's workstation screen) are updated as new DQL results are created.

Figure 1 shows Queries and Results tasks running in a workstation; as each new query is defined or result is requested, these windows are updated. EASE allows the user to browse through the windows holding query results with standard mouse pointers and pop-up menus. EASE maintains connections between windows, so a user can issue commands to one window by running a command in another window.

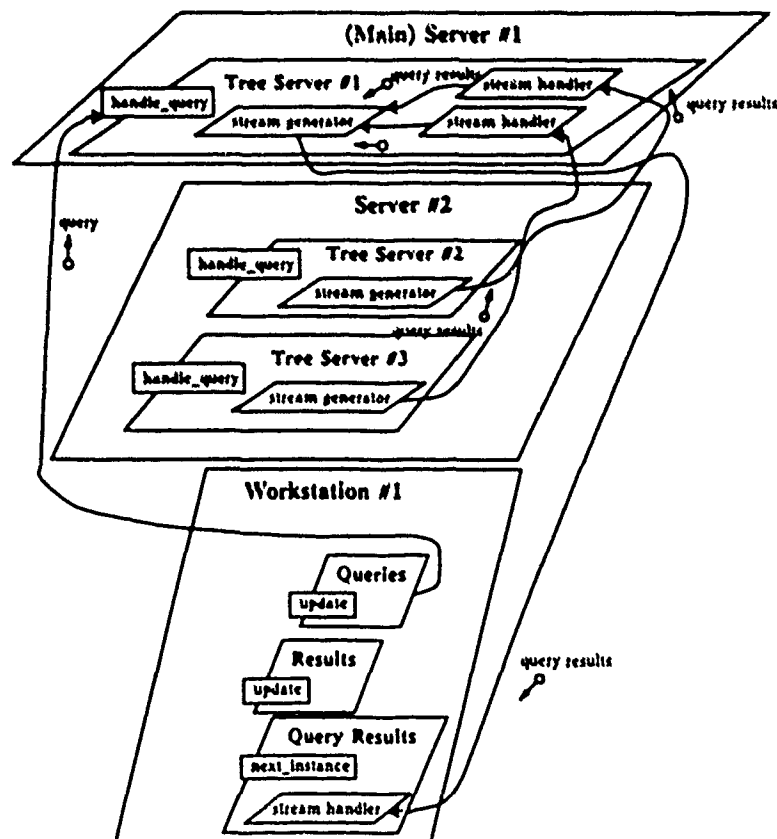


Figure 1. Query Involving Multiple Servers

7.4 PORTABILITY TO OTHER WORKSTATION ENVIRONMENTS

The current DQL (and EASE) implementation runs on Sun workstations using the Sun Visual/Integrated Environment for Workstations (SunVIEW®). Both EASE and DQL would be available to a broader class of user if they ran under a workstation environment that was an emerging industry standard. Both the 'X' and Sun Network extensible Windowing System (SunNeWS®) environments would be possible hosts for future versions of the workstation interface tools.

8 FUTURE WORK

8.1 CURRENT STATUS

DQL is being implemented in phases. The initial phase consisted of three parts: a retooling of the ACS front end to Sun servers as a source for DIANA trees; creating a DQL query language processor that converts a query (in ASCII text) into a semantically correct set of calls to a DIANA tree server; and implementing a DIANA tree server for the creation and management of subqueries and intermediate results. Later phases will provide a - rerouting system for the creation of queries and full support of queries which cross server boundaries. Each phase will also provide standard queries that Ada analysts can use as an introduction to browsing through Ada software.

8.2 DQL AS A LAYER FOR FUTURE TOOLS

The EASE environment is intended as a layer on which tools such as DQL could be built. DQL's tools provides a variety of interfaces that other tools can use to provide their Ada information. Examples of these interfaces include the DIANA tree server, the DQL query parsers, and the scrollable DQL query result windows.

An example of a tool that could use DQL is an expert system that needs information about Ada source to populate its knowledge base. Sometimes proper checking of a design rule may involve complex forward chaining rules or complicated pattern recognition. Such design rules could be captured in an expert system, with RPC calls to DIANA servers used as part of the inferencing mechanism.

8.3 CONVERSIONS OF EXISTING TOOLS

Existing tools will be modified to take advantage of the information that can be provided by DQL queries. For example, the Carleton Embedded system Design Environment (CAEDE) Buhr diagram editing tools (Buhr86), running on Sun workstations, could be modified so diagrams could be created from Ada source code as well as CAEDE currently creates the source code from the Buhr diagrams. This would be done by querying to determine library units and the tasks within them, and then seeing how the units call each other. Such modified tools would allow the Ada source's design to be presented in a form (such as the Buhr notation) that is familiar to a particular analyst.

® SunVIEW and SunNeWS are trademarks of Sun Microsystems, Inc.

ACKNOWLEDGEMENT

The author wishes to thank David Emery for his work in getting the tree servers to work. The author also wishes to thank Marlene Hazle, Richard Hillard, and Steven Livitchouk for their support and comments during the development of DQL.

REFERENCES

- (Bach87) Bachman, B., "Design Automation for Ada Development Under DOD-STD-2167," *Proceedings of the 1987 ACM SIGAda International Conference on the Ada Programming Language*, ACM Press, 1987.
- (Buhr84) Buhr, R. J. A., *System Design with Ada*, Prentice-Hall, 1984.
- (Buhr86) Buhr, R. J. A., et. al., *CAEDE 1.5 User's Guide*, Technical Report No. SCE-86-11, Carleton University, 1986.
- (Byrn88) Byrnes, C., "ESD Acquisition Support Environment (EASE)," *Proceedings of the Sixth National Conference on Ada Technology*, 1988.
- (Chen76) Chen, P., "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems*, Vol. 1, No. 1, Mar. 1976.
- (Cher86) Cherry, G. W., *PAMELA Designer's Handbook*, Thought® Tools, Inc., 1986.
- (Date82) Date, C. J., *An Introduction to Database Systems*, Addison-Wesley, 1982.
- (DOD83) Department of Defense, Ada Joint Program Office, *Ada Language Reference Manual*, ANSI/MIL-STD-1815A-1983, 1983.
- (Evan83) Evans, A., and K. J. Butler, *Descriptive Intermediate Attributed Notation for Ada Reference Manual*, TL-83-4, Tartan Lab, 1983.
- (Gord83) Gordon, M., "The Byron Program Development Language," *Journal of PASCAL and Ada*, June 1983.
- (Hamm81) Hammer, M. and D. McLeod, "Database Description with SDM: A Semantic Database Model," *ACM Transactions on Database Systems*, Vol. 6, No. 3, Sept. 1981.
- (Hoar85) Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall, 1985.
- (Inte86) Intermetrics, Inc., *ACS Compiler System User's Manual*, IR-MA-764, 1986.

- (Luck84) Luckham, D. C., et al, *ANNA: A Language for Annotating Ada Programs*, Technical Report No. 84-261, Stanford University, 1984.
- (Luck87) Luckham, D. C., et al, *Task Sequencing Language for Specifying Distributed Ada Systems*, Technical Report No. CSL-TR-87-334, Stanford University, 1987.
- (Niu84) Niu, J. and P. Wallis, *Portability and Style in Ada*, Cambridge University Press, 1984.
- (Perk87) Perkins, J. A. and R. S. Gorsela, "Experience Using an Automated Metrics Framework to Improve the Quality of Ada Software," *Proceedings of the Fifth National Conference on Ada Technology*, 1987.
- (Repi84) Repi, T. and T. Teitelbaum, "The Synthesizer Generator," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984.
- (Soft82) SoftTech, Inc., *Ada Software Design Methods Formulation: Case Studies Report*, United States Army Communications Electronics Command, 1982.
- (Ward85) Ward, P. T. and S. J. Mellor, *Structured Development for Real-Time Systems*, Youdon Press, 1985.

BIOGRAPHY

Christopher Byrnes is a Member of the Technical Staff of the Software Center of The MITRE Corporation. He received his B.A. from Tufts University in 1976, his M.S. from the University of Lowell in 1980, and his M.S.E. from the Wang Institute of Graduate Studies in 1984. His interests include software development methods, analysis of design products, and the Ada programming language. His mailing address is: The MITRE Corporation, Burlington Road, N/S A156, Bedford, Mass. 01730. He can also be reached at cb@mitre.org (InterNet) and at ...!decvax!linus!mbunix!cb (UUCP).



ADA PORTABILITY AMONG HETEROGENEOUS SYSTEMS

NASSER BAZZI and BENJAMIN CASALO

Advanced Software Technology, CECOM

ABSTRACT

Most government contracts are developed by many contractors who often utilize their own resources to perform their assigned task. Compatibility becomes a problem at the time of integration if different resources have been used. Therefore, engineers, or possibly consultants, must be utilized to solve the incompatibility problem. This process is not only inefficient but often costly to both the company and the government in terms of money and time.

Designing an Ada utility package, which is independent of the operating system, is a feasible solution to the portability problem since changes to the source code are now going to be carried out by the utility rather than the user. This is accomplished by using fixed specifications along with one of several bodies. The body chosen depends directly on the operating system in use.

INTRODUCTION

Heterogeneous systems are incompatible systems because of dissimilarities in both their properties and characteristics. These differences are most evident in their keyboard interfaces, file management system support functions and I/O routines, and communications.

An Ada source program, which is compiled using a particular compiler and operating system, will not be able to run under a different operating system, even if the same compiler is used, unless changes are made to the source code. Making these changes to the source code is undesirable, especially if the program has to run on many different systems, since this process is both time consuming and expensive.

In their book, *Portability and Style in Ada* [1], John Nissen and Peter Wallis, state the fact that "A perfectly portable Ada program would, without any change, be compilable by any valid Ada compiler". Furthermore, they proposed the equation below to measure the portability of any given Ada code.

$$1 - \frac{\text{(Cost of re-implementation on the new target)}}{\text{(Cost of original implementation)}}$$

If there is no re-implementation cost, the formula will yield the original implementation cost.

An intermediate layer which helps in creating different images out of the same source code, depending on the environment, relieves the user from having to make the changes to the source code himself. This will set the re-implementation cost to zero and the cost of the development will not be affected by any overhead due to the new code. The layer, which sits between the user's program and the operating system, will have to perform the operating system dependent calls. In addition to its needed routines, the layer can expand the capabilities of the language by including other user needed routines.

Since a concept rather than a tool is being proposed, the File Management System (FMS) layer is being focused upon, as an example, since the same procedure can be followed for the other layers.

After designing and implementing this project, a driver was written to test all of the operating system's dependent features on the layer. This driver was successfully compiled and executed in both DOS and UNIX, without requiring any changes or modifications.

APPROACH

The following steps were followed to create the intermediate layer. As stated in Figure 1, the layer can be broken into three major sections according to the types of services each section is to provide.

Section 1, the Communications Section, will allow an Ada file to be transferred between different machines. It may also add the features needed for the compilation of code on a distributed system.

Section 2, the File Management System and I/O Section, will provide the interfaces from the user's program to the lower Input and Output application calls of the operating system.

Section 3, the Keyboard Interface Section, will provide a compatible input interface regardless of the system's keyboard.

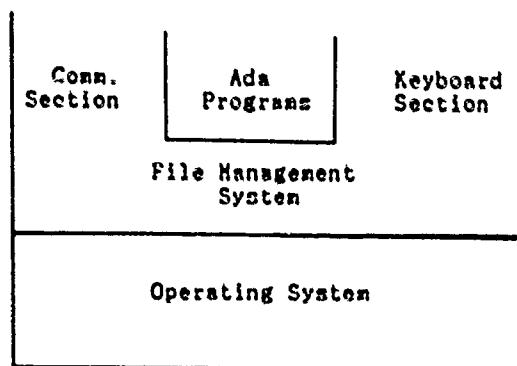


Figure 1

As stated in the introduction only the FMS layer will be developed completely in this exercise. The operating systems used were Unix and DOS.

The FMS layer can be represented, using the Ada language as the designing tool, as an Ada package.

This package will contain a set of specifications that is common to both of the operating systems in use. The specifications were designed using the Unix operating system semantics as the building block. Unix was chosen rather than DOS simply because the government prefers that operating system.

DESIGN

The specifications for the FMS package were based on the text_io package specifications from the Ada Language Reference Manual [2]. In addition to the routines already contained in text_io, additional routines were added to the new package specifications in order to handle directory related calls. The result of this was a package called Portable_Text_io.

In order to facilitate the use of the new package, it was imperative that the Portable_Text_io package ran exactly as did the text_io package. Generic instantiations of the generic packages inside the text_io specifications were also included in Portable_Text_io. In order to accomplish this abstract view the specifications of the generic modules in the new package included a type that was generic not only to the module being developed, but also to the module inside text_io.

After the set of specifications was developed, two package bodies, one for the DOS calls and one for the Unix calls, were developed. The package body that handled the DOS dependent calls contained the pragma interface to DOS to perform the requested call. On the other hand, the Unix dependent calls were handled by routines written in C, and utilized pragma interface to the C language.

In Figure 2 we can see how the call generated at the user layer will propagate to the operating system layer.

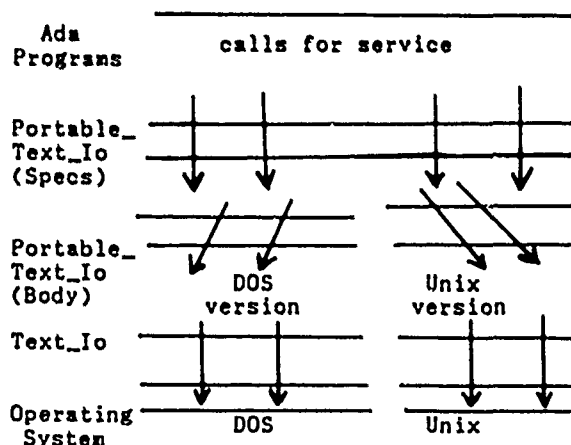


Figure 2

CONCLUSION

The Department of Defense is now, more than ever, requiring contractors to use the Ada programming language to develop their contracts. In order for the contractor to fully satisfy the required specifications of the contract, he will assign different parts of the contract to various developers. These developers in turn may use different systems to complete their individual task. Once the separate parts are completed, the next step is to integrate them on to the target computer.

The use of the Portable_Text_Io package will allow the user to run his system on two different environments, namely the IBM PC-AT and the SUN3 workstation.

The purpose of this project was to set an example of how to develop and implement such a layer. This was done by using two characteristically different machines, the IBM PC-AT and the SUN 3 workstation, running DOS and Unix respectively.

ACKNOWLEDGEMENTS

Special thanks to Dr. Thomas J. Wheeler for sharing his knowledge and expertise concerning this concept. Thank to Mr. Lee Van Lam who was one of the original developers of the Thesis Work at Monmouth College, and to Miss Karin Cookley for her contribution editing the paper.

REFERENCES

- [1] John Nissen and Peter Wallis. Portability and style in Ada. Cambridge University Press, 1984.
- [2] Ansi / Mil-Std 1815A. Language Reference Manual
- [3] J.G.P. Barnes Programming in Ada Addison-Wesley Publishing Company
- [4] Putnam P. Texel Introductory Ada Wadsworth Publishing Company
- [5] Richard Wiener and Richard Sincovec Software Engineering with Modula-2 and Ada. John Wiley and Sons.
- [6] Edmund R. Matthews "Observations on the Portability of Ada I/O" Ada Letters, Vol. VII Num. 5 Sept. 1987

Ada Compiler Validation: Purpose and Practice

Rosa Williams and Phil Brashear
SoftTech, Inc., Fairborn OH

Steve Wilson
Wright-Patterson AFB OH

if we assume that validation guarantees that compilers adhere strictly to all the syntactic and semantic specifications of the LRM.

Abstract

The real meaning of the term "validated Ada compiler" is often misunderstood. One common inference is that the validated compiler is bug-free; a second is that the compiler behaves precisely as specified by the Ada Language Reference Manual (LRM); still another perception is that a validated compiler supports every feature described in the LRM and is judged to be a "good" compiler. By examining the validation process and the test suite on which validation is based, we hope to correct some of these misunderstandings.

In other cases, the LRM fails to completely specify some semantic behavior without explicitly saying so. For example, LRM 5.5/6 says that the parameter of a FOR loop "is an object whose type is the base type of the discrete range." If a loop parameter is used in a CASE statement, it is important to know the subtype of the parameter (in particular, whether the subtype is static). To illustrate this, consider the following code:

```
for INDEX in 1 .. 100 loop
  case INDEX is
    when 1..50 => ... ;
    when 51 .. 100 => ... ;
  end case;
end loop;
```

If the subtype of INDEX is the static subtype 1 .. 100, then the CASE statement is legal; but if the subtype of INDEX is the base type, Integer, then the CASE statement is illegal because no OTHERS clause is given. Which is correct? The LRM does not completely specify the semantics of the situation, so, again, two validated compilers could produce different behavior (one rejecting the program, the other accepting it).

(Actually, in the case of the loop parameter, the standard is now interpreted to specify that the subtype of INDEX is determined by the discrete range. The case is covered by AI-00006, a Binding Interpretation of the standard, approved by the International Standards Organization's Working Group 9 (WG9) and the Ada Joint Program Office (AJPO) in July of 1986.)

The Unattainable Goal Of Validation

The theoretical goal of Ada compiler validation is to ensure that a translation system purchased as an Ada compiler obeys the syntax and semantics specified by ANSI-MIL-STD-1815A, Reference Manual for the Ada Programming Language (LRM). If we assume that this goal is achieved, then we would expect that an Ada program would exhibit the same behavior when processed by any validated compiler and executed on that compiler's target system. However, there are at least two points on which this expectation fails.

Incomplete Specification

In some cases, the LRM explicitly leaves choices to the implementation, as in LRM 3.6.1/11: "For the elaboration of an index constraint, the discrete ranges are evaluated in some order that is not defined by the language." If the bounds of the discrete ranges in an index constraint are given by functions with side effects, then the LRM does not completely specify the effect of the program. It is quite likely that two validated compilers would produce code whose execution would exhibit different behavior in this situation, even

Incomplete Testing

Even if the standard were perfect, it would not be possible to verify absolute conformity. It is well known that, for a program with any sizable data space, exhaustive testing of the program is impractical. In the case of a compiler, the data space is the set of all collections of text files that are not too large to be processed by the compiler. To guarantee absolute adherence to a standard would require that every such collection be submitted to the compiler, with the criteria

that every collection not representing a legal program be rejected and that every collection representing a legal program exhibit the expected behavior. Such exhaustive testing is simply not feasible.

The Reality of Validation

In reality, Ada compiler validation depends on the completeness and correctness of the Ada Compiler Validation Capability (ACVC). This set of test programs and support software has evolved with the Ada effort. It is not complete (in the exhaustive sense), nor will it ever be complete. The current version, ACVC 1.10, contains over 3700 test programs, but important areas of the language are not yet tested. Thus, an Ada compiler can be tested and validated without correctly supporting the very feature that a particular program might depend on.

In addition to its being forever incomplete, the ACVC will always be subject to error. The tests are written carefully, with constant review, but there are many subtle points of the language that do not come to light until someone tries to use them. For example, a test may make assumptions that are valid for every known implementation because of the existence of standard implementation techniques. Yet, an implementer who is an independent thinker may use a non-standard technique that is permitted by the standard, but that has a totally unexpected impact on the test's behavior. This situation happens frequently, and will continue to do so as compiler technology becomes more sophisticated.

Thus, the fact that an Ada compiler is validated does not guarantee that it adheres precisely to the standard, for such a guarantee is impossible. It most certainly does not guarantee that the compiler contains no "bugs," for the test suite is not designed for debugging (although implementers do often find bugs when running the ACVC). Validation does not ensure that compilers are efficient, either in terms of time or memory usage, for there are no tests for efficiency in the suite. It does not guarantee that large, complex programs can be handled correctly, for the test suite consists of many small programs, designed to test specific language features.

What, then, is the meaning of validation? What can we assume about a validated Ada compiler? First, a validated Ada compiler has correctly processed the most widely portable body of Ada software in existence. Second, it has done so under the supervision of an impartial validation team, and the validation report produced by that team has been scrutinized by the vendor and by another impartial validation agency. Third, any behavior not strictly in accordance with the expectations of the test suite has been ruled justifiable by the validation agencies and has been thoroughly documented in the validation

report. Finally, the implementer has attested that no extensions to the language have been knowingly implemented. These characteristics of an Ada compiler should give the Ada user assurance that the best possible effort has been made, by all parties, to see that the compiler conforms as closely as possible to the Ada language standard.

The Validation Process

The purpose, in theory, of Ada compiler validation testing is to verify the conformity of an implementation with the Standard, MIL-Standard 1815A. As may be apparent from this stated purpose, "compiler" validation testing is actually a misnomer. In fact, the entire implementation, including the compiler and the host and target computers and operating systems, is really tested because a change to any part of the implementation can affect compilation results. According to MIL-STD-1815A, also known as the Ada Language Reference Manual (LRM), an implementation conforms to the Standard if and only if it satisfies each of the following six requirements:

1. It correctly translates and executes legal Ada program units that do not exceed the capacity of the implementation.
2. It rejects all program units that exceed the implementation's capacity.
3. It rejects all program units containing errors that the LRM requires are to be detected.
4. It supplies all predefined program units that the Standard requires.
5. It contains no variations except as allowed by the Standard.
6. It specifies variations permitted by the Standard.

A true all-inclusive conformity verification needs to cover all six requirements and thus ensure that the implementation is neither a "subset" nor a "superset" of the Ada language.

It is impractical, if not impossible, to verify an implementation's conformity to the Standard. For example, to check that the implementation rejects all program units containing errors would require that all possible errors be identified and a test be written for each one, an endless exercise.

What can and has been done, however, is to develop a measuring stick of conformity. This measuring stick is called the Ada Compiler Validation Capability (ACVC). As a first step in developing the ACVC test suite, the LRM was broken into discrete test objectives. The resulting document was the ACVC Implementers' Guide (AIG).

To obtain a base validated compiler, an implementer must contract with one of five Ada Validation Facilities (AVFs) to perform the validation. The implementer obtains a copy of the ACVC test suite.

Prevalidation

Once the implementer has run the ACVC tests on his implementation and believes that he has a complete and correct set of test results, he submits these results, called prevalidation results, to his AVF for analysis. If the implementer finds tests he believes incorrectly showed that the compiler failed these tests, he submits arguments to the AVF disputing those tests. The AVF forwards the arguments to the Ada Validation Organization (AVO) which handles test disputes sent them from all AVFs. For each test, the AVO decides either that the implementer must change his compiler to pass the test or that the test may be declared not applicable for the subject implementation. When the AVF has received from the implementer all ACVC test results and has graded each test as either "passed" or "not applicable", prevalidation is complete. Failed tests must be resolved before prevalidation is considered complete.

On Site Testing

Once prevalidation is completed, an AVF test team travels to the implementer's site on a prearranged date and takes with them the ACVC test suite, customized for the implementation under test. On site, the test team loads the ACVC onto the implementation under test and verifies that on-site results match the previous prevalidation results.

Once the implementer has successfully completed on-site testing and has signed a Declaration of Conformance, affirming compliance to the LRM as measured by the ACVC, the AVF sends a notice of completion of on-site testing to the AVO. When the AVO determines that the validation attempt is successful, they direct that a Validation Certificate, listing both the implementer and implementation, be sent to the validation customer. A copy of the validation certificate is maintained by the AVO and serves as the permanent validation record.

The Validation Summary Report

Following on-site testing, the AVF prepares a Validation Summary Report (VSR) which describes the implementation tested, the detailed procedures used in the validation, and the tests declared not applicable. All compiler switch settings used in the validation are recorded in the VSR. One purpose of the VSR is to allow the validation to be completely reproduced. This capability could become extremely important if a compiler's conformance is challenged.

The VSR highlights several validation shortcomings. An implementation is tested for a specific compiler, host, target, operating system, and set of switch settings including optimization. It is quite possible for ACVC test results to differ, if even one component of the implementation is only slightly changed.

An implementer typically validates under one set of switch settings and sells the compiler, as a validated Ada compiler, under a different set -- one that provides better performance. But the compiler cannot necessarily be expected to pass the same set of ACVC tests when the switches have been changed. The dilemma has been discussed by the Ada Certification Body and to some extent resolved. The Validation Certificate does not, due to space limitations, list all switch settings used in the validation. Only the VSR lists the validated switch settings used in the validation and for which the ACVC test results apply. Thus, the VSR serves as the authoritative and complete validation documentation supporting the validation certificate.

Derived Implementations

Implementers have effectively argued that to require that a base-validation be performed for every possible compiler on every possible host/target configuration is both impractical and unnecessary. To validate for every variation of host, target, compiler maintenance update and every combination of all three would create more base-validations than could reasonably be expected to be completed. Furthermore, there exist many implementations slightly different than a base-validated implementation that would reasonably be expected to conform exactly as the base does. For example, it is completely reasonable to expect that a compiler will behave similarly on any VAX within the entire VAX family, and there are many minor upgrades to compilers to improve performance that do not in any way affect conformance.

To accommodate those instances when an implementation is so similar to a base-validated implementation that it would reasonably be expected to perform exactly as the base, the Certification Body has permitted such implementations to be registered as derived compilers. Derived compilers are validated compilers. The basis for the validation status lies only with the compiler's documented similarity to a base-validated compiler. To register an implementation as a derived implementation, an implementer submits a request to the AVF that performed the base-validation. The implementer provides his rationale for the derivation, documentation supporting that rationale, and a signed Declaration of Conformance listing the base and candidate derived configurations. The AVF reviews the rationale and supporting documentation and, if derivability seems plausible, recommends to the AVO that the derivation be accepted. When the AVO concurs with the opinion of the AVF, they direct that the now-

derived compiler be added to the validated compiler list.

Users should be aware that, although a derived implementation is validated, it has not been tested against the ACVC by an independent test organization. Consequently, the conformance depends to a much greater degree than a base-validation on the affirmation of the compiler implementer.

Conformance Testing

Validation deals only with conformance testing. It does not test the efficiency or the performance of the compiler. Since efficiency and performance are important in evaluating the utility of a compiler, validation cannot possibly indicate how "good" the compiler is, where "good" refers to the compiler's performance and efficiency.

Validation is limited in its test of conformity by practical limitations and procedures established by the Ada Certification Body and by limitations of the ACVC itself. The ACVC is a measure and not a litmus test of the conformity of an Ada compiler. To understand both the value and limitations of the ACVC, it is important to know how the test suite is structured.

The Ada Compiler Validation Capability (ACVC)

The ACVC is a changing body of tests. Up until now, a new version has been released every year. As of the writing of this paper, the current version is ACVC 1.10 which was released as a pre-release version on 1 December 1987; it was released as a final version on 1 May 1988 and became the official version for use in validations on 1 June 1988.

The tests in the suite are based upon the LRM, as interpreted by the ACVC Implementers' Guide (AIG). Basically, the AIG follows the chapter, section, and subsection structure and numbering of the LRM. The AIG lists test objectives for each subsection. Each objective is designed to cover one atomic feature of the Ada Language. The tests are also written atomically; one language feature is covered per test.

Each of the tests in the suite is a short Ada language program. Most of these are executable, and, if executed properly, will write the test name followed by the word "PASSED" to standard output. There are, however, tests which are not meant to execute. These test programs contain intentional semantic or syntactic errors and were written for the purpose of determining whether a compiler can detect these errors.

Classes of Tests

The tests of the ACVC are divided into six classes, A tests, B tests, C tests, D tests, E tests, and L tests. Class A tests check that a compiler does accept certain legal Ada language features. For example, there is an A test which checks that compilers accept an enumeration type definition which contains a single enumeration literal. Class B tests check that compilers reject constructs which are not legal Ada language features. For example, there is a B test which checks that a compiler rejects an enumeration type definition which consists of empty parentheses, i.e., which contains no enumeration literals. Class C test check that a compiler not only accepts legal Ada code, but also that the code is executed correctly. For instance, a C test might check that not only are logical operators for arrays of Boolean elements accepted, but also that such operations yield a correct result. Class D tests check compiler capability. There is a D test which determines the number of nested block statements that a compiler can support.

A compiler's performance on A, C, and D tests is usually judged by whether the successful compilation, linking, and execution of the test results causes a message containing the word "PASSED" to be printed. For class B tests, a compiler's performance is judged by the compiler's finding an error at those places indicated in the test. Class E is for tests the performance of which cannot be judged by either of the above methods. For example, some class E tests determine whether pragma LIST and pragma PAGE behave correctly. These tests must not only compile, link, and execute correctly, but must also produce correct listing files. The final class, class L, consists of tests that should compile correctly but which must fail at link time. An example of such a test would be one which consists of a main program for which a necessary subunit is missing from the program library.

By convention, the name of a test indicates the class of the test and the chapter, section, and subsection of the LRM to which the test pertains, and the number of the test objective in the AIG. For example, if the name of a test is "A35101B", then the first character indicates that this is a class A test. The second through fourth characters indicates that the tests covers an objective taken from subsection 3.5.1 of the LRM. The last three characters indicate that the test is based upon the second part (part B) of the first test objective in that subsection.

Version 1.10 of the ACVC

Version 1.10 of the test suite consists of 3717 tests, an increase of 621 tests over version 1.9. The major portion of new tests covers issues from Chapter 13 of the LRM, "Representation Clauses and Implementation Dependent Features." In writing these tests the attempt was made to

include not only those constructs which might be almost universally implemented, but also constructs which, although supported by the language, are not usually supported by implementers. An example of the former is a test which might make use of a size clause, specifying an a size equal to `INTEGER'SIZE` divided by two and applied to a small integer type. Examples of the latter can be found in tests which provide address specification clauses for subprograms and task units. Another example of the latter can be found in tests which first declare a floating point type, `FLOAT5`, with precision five and then specify that the size of `FLOAT1`, a floating point type of precision one, should be `FLOAT5'SIZE` divided by two.

Much of the controversy surrounding the Chapter 13 tests stems from the fact that up until now, little or no effort has been made to establish a uniform interpretation of the issues in Chapter 13. Although the tests in the suite are not intended to resolve the issues, the tests have caused many questions to be raised. For instance, concerning the size of a type the LRM clearly states: "X'SIZE ... Applied to a type or subtype, yields the minimum number of bits that is needed by the implementation to hold any possible object of this type or subtype." If `BOOLEAN'SIZE` for a given implementation is one bit, does this mean that any object of type `BOOLEAN` can fit into one bit? It seems as if the answer to this question should be "YES." Yet, it is not uncommon for an implementation which reports a size of one bit for type `BOOLEAN` to require four bits or eight bits to hold a `BOOLEAN` object which is an array or a record component. If B is an object of type `BOOLEAN`, then what is the relationship between `B'SIZE` and `BOOLEAN'SIZE`? Is there necessarily any relationship?

Another item upon which there is some disagreement is the relationship between a `STORAGE_SIZE` clause and the `STORAGE_SIZE` attribute. If a `STORAGE_SIZE` clause specifies a collection size of 1024 storage units for an access type T, then can an implementation legitimately reserve more space for the collection than specified? Furthermore, what value should be returned by the attribute `T'SORAGE_SIZE`? If more space is reserved than the amount specified, then must the `T'SORAGE_SIZE` attribute return the actual amount of space reserved, or should the attribute merely return the number of units specified? Does it matter what value the attribute returns? Of what use is the value returned by the attribute? Should an implementation be judged to be in error if the value of the attribute does not reflect exactly the amount of storage reserved? If no representation clause is given for T, then although a compiler cannot reject the expression `T'SORAGE_SIZE`, what value does this expression have?

No discussion of Chapter 13 issues would be complete without a word about addresses and address clauses. It appears that few, if any, implementations support address clauses for subprograms, tasks, or packages. However, is it totally impractical for an implementation to support such address clauses for program units? If so, then why are such constructs supported by

the language? Although it is legal to put an address clause for an object declared inside of a subprogram or even inside of a nested subprogram, does it really make sense to refer to the address of such a local object? Can an implementation legitimately return the same value (a value of zero, for example) for all label names and block names? As of today, most of the issues involving Chapter 13 tests are yet to be resolved. The debate continues.

Untestable Objectives

Although the goal of the ACVC is to provide as thorough coverage of the language as possible, there are some tests which implementers should not expect to see in any of the upcoming versions. Most of these may be found in Chapter 14 of the LRM. For example, there are no tests upcoming which involve low level input and output. Since the procedures (including the parameters) specified in the package `LOW_LEVEL_IO` are left totally up to the implementation, there is no practical way to ascertain that they behave correctly. Another test that is not forthcoming is a test that checks that the exception `DEVICE_ERROR` is raised as specified by the LRM, i.e., when there is a malfunction of the underlying system. The writing of this test is awaiting the discovery of a harmless way to cause a system to malfunction.

Future of the ACVC

The future of the ACVC is unclear at this time. Version 1.11 is in the development stage. It will be in many ways like version 1.10. The major blocks of new tests will come from Chapter 8 of the LRM (tests dealing with visibility and renaming), Chapter 13 (tests dealing with more of the same issues as those Chapter 13 tests in version 1.10), and from Chapter 4 (tests dealing mainly with type conversions and with real arithmetic). What happens to the suite after version 1.11 is still "up in the air." One proposal is that the suite be "frozen" at version 1.11 and that although revisions to existing tests will be allowed, there will be no new tests written. Another idea is to produce a version 1.12 of the suite in the same way as previous versions have been produced. A third proposal is that the philosophy behind the ACVC and its use be changed. A new suite of tests would be produced in which each test will check combinations of features rather than a single feature. Whether the test suite will take one of these three directions or some other direction must await the decision of the Ada Maintenance Organization (AMO) with the approval of the Ada Joint Program Office (AJPO).

References

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

About the Authors



Rosa Williams received the A.B. in Mathematics from Spelman College, Atlanta, Georgia and the M.S. in Computer Science from Wright State University, Dayton, Ohio. Presently, she is employed by SofTech, Inc. where she serves as the Senior Test Reviewer for the ACVC test suite.



Phil Brashear received the A.B. in Education/Mathematics from the University of Kentucky in 1962 and the M.A. in Mathematics from Northwestern University (Evanston, Illinois) in 1965. Presently, he serves as the Ada language expert on the SofTech, Inc. contract to operate the Ada Validation Facility at Wright-Patterson Air Force Base. His duties include analysis of Ada validation results and managing the ACVC maintenance/development effort. From 1965 through 1985 he was a member of the faculty of the Department of Mathematics, Statistics, and Computer Science at Eastern Kentucky University.



Steve Wilson received his Bachelor of Science Degree in Mathematics from South Dakota State University in May 72. He later received a Masters of Science Degree in Mathematics from the University of Nebraska in Lincoln in August 74.

Mr. Wilson worked as an Associate Systems Analyst for Burroughs Corp. in Detroit in 1975. In December 75, he began work as a data analyst for aircraft flight testing at the 4956th Test Wing, FFA, Wright-Patterson AFB, Ohio. In August 1986, Mr. Wilson joined the Ada program as the Ada Task Leader at ASD/SCEL, Wright-Patterson AFB. He is currently the Technical Director of both the Ada Validation Facility and the ACVC Maintenance Organization.

Rosa Williams and Phil Brashear may be contacted at:

SofTech, Inc.
3100 Presidential Drive
Fairborn, OH
45324-2039

Steve Wilson may be contacted at:

Aeronautical Systems Division
SCEL
Building 676, Area B
Wright-Patterson AFB OH 45433

How to Live with TEXT_IO

by Do-While Jones

Abstract

TEXT_IO is the standard Ada package for input and output of character data. It is commonly used to transfer data between devices and files. Unfortunately, its specification is inconsistent and loose enough that vendors have implemented it differently, resulting in portability problems and surprising quirks. These surprises make IO annoying and frustrating to most Ada beginners, and even to a few seasoned veterans. To make matters worse, TEXT_IO was designed to be a file interface, but it is often pressed into service as a user interface. It doesn't do this job very well, so application programs that use TEXT_IO for a user interface tend to make users unhappy.

Introduction

This paper describes some of the portability problems you are likely to have if you use TEXT_IO. It tells why a file written by TEXT_IO on one machine might not be read correctly on a second machine, and why a program that works properly on one machine puts a blank line between user prompts (or writes prompts on top of each other) when transferred to another machine.

You will also find out why quirks in TEXT_IO cause some programs to seem to skip over user inputs without processing them. I'll show you how to write numbers and enumeration types in ASCII format without instantiating a generic IO package. Finally, I'll suggest some other user interfaces that eliminate the need for TEXT_IO entirely.

Problems with TEXT_IO

File Portability Problems

Suppose you have a data file (containing numbers, not text) that you want to transfer to a second computer. You know better than to try to transfer binary files. The two machines might use a different number of bits to represent integers, so each integer written by a 32-bit machine would get "unpacked" into two

integers on a 16-bit machine. Even if both machines use the same number of bytes per integer, one might store the high byte first while another might store the low byte first. Floating point numbers are even less portable because there are so many different ways to represent them. (VAX/VMS¹ uses 4 different internal forms for real numbers.) Different computers generally use different numbers of bits for the mantissa and exponent. You are asking for trouble if you try to transfer files in binary format.

You might think you can avoid all those problems by using TEXT_IO to convert the numbers to character strings in a text file, and then transfer the text file from one machine to the other. Well, it's not that simple. You may discover that files written by one machine will raise CONSTRAINT_ERROR or DATA_ERROR when read on another machine. That's not so bad, because at least you know there is a problem. Sometimes your data will be skewed forward or backward one location in the file, causing the data to be read into the wrong variables. (That is, the value for the third element of an array may end up in the second or fourth element.) When this happens, there may not be an error message.

User Interface Deficiencies

TEXT_IO makes a terrible user interface. It treats the user's terminal just like a file and lacks features that humans need. Files never make mistakes, so they don't need a rub out key. Files never enter passwords which shouldn't be echoed to the screen. Files never want to insert or delete text. Files never need help, or want to enter the default response. Files never want to clear a screen or move a cursor. Files never realize the program has run amok and try to send an unsolicited CTRL-C to stop the process. Files never want to press a special function key. Users often want to do all these things, but TEXT_IO won't let them because it wasn't designed to support people.

User Interface Surprises

The `Get` and `Get_Line` procedures don't work the way most people seem to expect them to. How many people have tried to use `TEXT_IO.Get(C : character)` to try to build a line editor, only to discover that no matter what you do, it won't respond to a carriage return? How many people have written programs with a mixture of `Get` and `Get_Line` procedures that seemed to hang forever, or take data before the user entered it? Practically every Ada programmer, I bet.

When this happens, don't blame the compiler vendor. There's nothing wrong with the compiler. It's just conforming to the specification. You'll see why after we examine some of the strange passages in the Ada Language Reference Manual (LRM).

User Interface Portability

Since input data editing might not be done by the operating system service called by the `Get` procedure, you never can tell if `CTRL-X` will erase a whole line, or if backspace will be the same as delete. You might also discover that a program that runs fine on one system does strange things on another. The user prompts might appear on consecutive lines on the first machine, but may have blank lines between them on a second machine. Worse yet, the prompts might appear on top of each other on another machine.

Causes of TEXT_IO Problems

Incorrect Implementations

Six years ago, Ada pioneers had to use unvalidated, partial implementations. Those compilers were full of bugs. In those days, there were some IO errors because `TEXT_IO` wasn't implemented correctly. I haven't seen a problem that was the result of a `TEXT_IO` implementation error in the last few years, but I think there is still a tendency to blame the compiler whenever `TEXT_IO` doesn't work the way the programmer thinks it should. Even in those cases where a program runs differently with two different versions of `TEXT_IO`, you can't be certain either of them is wrong because the specification allows so many options.

Loose Specification

`TEXT_IO` leaves some important details unspecified. Here are two troublesome passages in the LRM:

"The actual nature of terminators is not defined by the language and hence depends on the implementation. Although terminators are recognized or generated by certain of the procedures that follow, they are not necessarily implemented as characters or as sequences of characters. Whether they are characters (and if so which ones) in any particular implementation need not concern a user who neither explicitly outputs nor explicitly inputs control characters. The effect of input or output of control characters (other than horizontal tabulation) is not defined by the language." LRM section 14.3 paragraph 7

"A page terminator is always skipped whenever the preceding line terminator is skipped. An implementation may represent the combination of these terminators by a single character, provided that it is properly recognized at input." LRM section 14.3.4 paragraph 51

In other words, there is no guarantee that two different implementations of `TEXT_IO` will use the same terminators. Therefore, the line-, page-, and file-terminators generated by machine 1 might be mistaken for data by machine 2 and generate `DATA_ERROR` exceptions. Perhaps the difference in terminators might not be detected at all, resulting in "off by one errors" (that is, reading item `N+1` when you think you are reading item `N`).

Here's a fictional example that illustrates what could happen: When Machine 1 writes a line, it writes "some string"<CR><LF>. The carriage-return/line-feed sequence is the line terminator for Machine 1. But Machine 2 might write a line with the line spacing first, so lines look like <LF>"some string"<CR>. It considers the line feed to be a character, and the carriage return alone to be the line terminator. Suppose you use these Ada statements to write a few lines to a file:

```

put_line(FILE,"A");
put_line(FILE,"B");
put(FILE,'X' & ASCII.CR & ASCII.LF);
put_line(FILE,"C");

```

If Machine 1 writes to FILE1.EXT and Machine 2 writes to FILE2.EXT, then the contents of those files will be as shown below. (Let <EOPn> and <EOFn> represent the end of page and end of file markers on Machine n, which might not be the same.)

FILE1.EXT	FILE2.EXT
A<CR><LF>	<LF>A<CR>
B<CR><LF>	<LF>B<CR>
X<CR><LF>	<LF>X<CR><LF>
C<CR><LF>	<LF>C<CR>
<EOP1>	<EOP2>
<EOF1>	<EOF2>

Suppose you have written an Ada program called *List* which uses *TEXT_IO* to list files. As long as FILE1.EXT remains on Machine 1 and FILE2.EXT remains on Machine 2, there isn't any problem. This is what you will see:

```
Machine_1>LIST FILE1.EXT
```

```

A
B
X
C

```

```
Machine_2>LIST FILE2.EXT
```

```

A
B
X
C

```

But suppose you transfer each file to the other machine. Here's what happens:

```
Machine_1>LIST FILE2.EXT
```

```

A
B
X

```

C
(Letter C might be immediately covered by an error message, caused by a bad end of page or end of file terminator. If not, it will probably be covered by the Machine_1 prompt.)

```
Machine_2>LIST FILE1.EXT
```

```

B
X
C

```

(Prompt or possible error message here.)

This example illustrates an annoying, but not critical, quirk. Suppose, however, FILE1.EXT and FILE2.EXT were data files. The first data item is A, but Machine 1 would think the first data item is a blank line, and Machine 2 would think the first item is B. This could result in *CONSTRAINT_ERROR*, or perhaps wrong answers without any error indication. If the page terminators or file terminators are different, it could raise *DATA_ERROR*. Failure to define standard terminators leaves the door open for all sorts of nasty portability problems.

Numeric Limits

A potential difference in line terminators isn't the only problem. Even if two machines use the same terminators, you can still run into trouble porting files containing ASCII representations of numbers. The string representation of a 32-bit integer may not be an allowable value for a 16-bit integer. The maximum AFT field sizes in *FLOAT_IO* and *FIXED_IO* might be different on two different machines, and the machine with the smaller AFT field might raise an exception if there are too many characters in that field.

To be honest, this isn't really a *TEXT_IO* problem, it is a machine capability problem. It only appears to be a *TEXT_IO* problem because you don't discover it until you try to use or instantiate the numeric IO packages in *TEXT_IO*. But if you achieve portability by using special numeric types that aren't derived from *Integer* or *Float* to get the same range and precision on any computer (for example, an array of digits), you won't be able to instantiate *TEXT_IO*'s generic packages to convert those variables to ASCII representations. Then it does become a *TEXT_IO* problem.

Confusing Inconsistency

Chapter 14 (which describes *TEXT_IO*) is probably the most confusing, inconsistent part of the LRM. When talking about the *Get* for characters it says:

"After skipping any line terminators and any page terminators, [Get] reads the next character from the specified input file and returns the value of this character in the out parameter ITEM." LRM section 14.3.6 paragraph 3

This says that every line terminator, no matter where it occurs, is skipped. Now, here is what it says about the Get procedure for strings:

"Determines the length of the given string and attempts that number of GET operations for successive characters of the string (in particular, no operation is performed if the string is null)." LRM section 14.3.6 paragraph 9

This implies it also skips all line terminators, because it calls the Get procedure for characters. A pattern is beginning to take shape. But wait, see what it says about integers (or real numbers):

"If the value of the parameter WIDTH is zero, skips any leading blanks, line terminators, or page terminators, then reads a plus or minus sign if present, then reads according to the syntax of an integer (or "a real") literal (which may be a boxed literal). If a nonzero value of WIDTH is supplied, then exactly WIDTH characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count." LRM section 14.3.7 paragraph 6 (or 14.3.8 paragraph 9)

So numeric forms of Get skip leading line terminators only if WIDTH is zero, and never skip a terminator that appears after a character of any kind has been encountered. Since integers are just a special kind of enumeration type, you might expect enumeration types to be similar. They aren't.

"After skipping any leading blanks, line terminators, or page terminators, reads an identifier according to the syntax of this lexical element (lower or upper case

equivalent), or a character literal according to the syntax of this lexical element (including apostrophes). Returns, in the parameter ITEM, the value of type ENUM that corresponds to the sequence input." LRM section 14.3.9 paragraph 6

What does it do about line terminators encountered after a significant character? It doesn't say. Suppose an object of a user-defined enumeration type can have the values HAND, AID, BANDAID, and BANDMASTER. How does TEXT_IO handle "HAND<terminator>AID", "BANDAID", and "HAND<terminator>MASTER"? I sure don't know.

So far, the specification has said, "Character Gets ignore all terminators; numeric Gets recognize terminating terminators, and sometimes ignore leading terminators; enumeration gets (other than integer Gets) always ignore leading terminators and might not recognize terminating terminators." Ah, if it was only that simple. But there's more.

Since all forms of Get for character data types ignore all terminators, we might expect Get_Line (which works only for strings of characters) would do the same. Not so.

"[Get_Line] Replaces successive characters of the specified string by successive characters read from the specified input file. Reading stops if the end of line is met, in which case the procedure SKIP_LINE is then called (in effect) with a spacing of one; reading also stops if the end of string is met. Characters not replaced are left undefined." LRM section 14.3.6 paragraph 13

This says Get_Line doesn't ignore any terminators (including leading ones), and skips over the terminator that causes input to cease. That's surprising because

"The character or line terminator that causes input to cease remains available for subsequent input." LRM section 14.3.5 paragraph 5

Is this a contradiction? Well, the title of section 14.3.5 is "Get and Put Procedures", so presumably the discussion in section 14.3.5 is limited to `Get` and `Put` and doesn't apply to `Get_Line`. But paragraphs 8 and 9 of that section specifically deal with `New_Line` and `Get_Line`, so one could argue that the title "Get and Put Procedures" is a general term that includes all output routines (including `New_Line`) and all input routines (including `Get_Line`).

Suppose there are seven characters before a line terminator, and you use `Get_Line` to fetch a 7-character string. Does it stop reading because it has read seven characters (and therefore not skip the line terminator)? or does it stop reading because there are no more characters on the line (and skip the terminator)? The LRM doesn't say. I expected it to skip the line terminator, but found it didn't on two validated compilers.

Operating System Dependencies

It is obvious that `TEXT_IO` was designed for files, not people. Files are record-oriented, and people are character-oriented. That's why the `Keystroke_Counter` program, shown in Figure 1, won't work.

Figure 1. A program plagued with problems.

```
with TEXT_IO; use TEXT_IO;
procedure Keystroke_Counter is
  KEYSTROKE : character;
  COUNTER   : integer;
  CONTROL_Z : constant character
    := character'VAL(26);
begin
  put_line("Press some keys, then"
    & " CONTROL-Z to quit.");
  COUNTER := 0;
  loop
    get(KEYSTROKE);
    exit when KEYSTROKE = CONTROL_Z;
    COUNTER := COUNTER+1;
  end loop;
  new_line;
  put("You pressed");
  put(integer'image(COUNTER));
  put_line(" keys.");
end Keystroke_Counter;
```

If someone gave you the listing of the `Keystroke_Counter` and asked you what it does, you would probably say that it counts the number keys pressed before the user presses `CONTROL_Z`, then prints the number of keys pressed. That's what it appears to do, but it doesn't.

The first problem is the `CONTROL_Z`. That's a special character that the operating system might filter out. It might be ignored, or it might cause the word `EXIT` to be printed on your screen (in reverse video) and immediately terminate your program. Since `CONTROL_Z` is likely to be the end-of-file terminator, `TEXT_IO` might notice that the `CONTROL_Z` isn't immediately preceded by a page terminator, and raise `END_ERROR`.

Suppose you try to avoid the problem by changing the exit line to exit when `KEYSTROKE = '.';`. It still won't work. Suppose you type three characters and then a period. The characters are echoed as you enter them. You type seven more characters, and then hit the carriage return. Suddenly it prints `You Pressed 3 keys`. That's because the operating system is treating the terminal like a record-oriented file. It waits for the end of record before it passes the complete string to the `Keystroke_Counter`. Then `Keystroke_Counter` examines the string one character at a time.

You might think you can solve the problem by changing the exit line to exit when `KEYSTROKE = ASCII.CR;`. Wrong again. `TEXT_IO` was designed to work with files. Files don't care about line terminators. They aren't limited to an 8.5 inch wide page, so they don't care how many characters are on a line. As far as they are concerned, line terminators are just meaningless symbols sprinkled about in the real data for cosmetic reasons, so the text will look nice on a printed page.

Remember, the character `Get` procedure ignores all terminators and throws them disdainfully on the ground. As we have already noted, the LRM doesn't specify what a line terminator is, but all the compilers I have used happened to pick the carriage return. So, if you try to use `get(KEYSTROKE)` to get a series of characters until a carriage return is entered, your program will never see the carriage return because it is discarded as a meaningless line terminator.

Solutions

Ad Hoc Solutions

The simplest (but not the best) solution is to find a unique solution to each unique problem. Consider the `TEXT_IO_Quirk` program in Figure 2. It is a nonsensical, contrived example to show what happens when you read two integers and a string. (It doesn't do anything with X or Y, and doesn't check to see if the user entered a response in lower case.)

Figure 2. A surprising quirk in `TEXT_IO`.

with `TEXT_IO`;
procedure `TEXT_IO_Quirk` is

```
package INT_IO is new
  TEXT_IO.INTEGER_IO(integer);
use TEXT_IO, INT_IO;

X, Y : integer;

RESPONSE : string(1..3);
LENGTH   : natural;

DONE : boolean;

begin
  new_line;
  DONE := FALSE;
  while not DONE loop
    put("Enter X: "); get(X); new_line;
    put("Enter Y: "); get(Y); new_line;
    put("Do you want to do it again?"
      & " (YES / NO) ");
    get_line(RESPONSE, LENGTH);
    if RESPONSE(1..3) = "YES" then
      DONE := TRUE;
    end if;
  end loop;
end TEXT_IO_Quirk;
```

If you compile and run the program, you will see that it prompts you to enter the integer X. After you enter a value, it prompts for Y. When you enter Y, it responds with both the prompts for YES/NO and X. It acts as if you entered a blank line instead of YES or NO. Since a blank line is not YES, it goes back to the top of the loop.

Why did it do this? After processing `Get(X)`, the line terminator was still "available for subsequent input" as section 14.3.5(5) requires. The `Get(Y)` procedure skipped it and read your second integer entry, stopping just before the line terminator. Then `Get_Line` read the line terminator associated with the entry of Y and thought you entered a null response to the question about whether to do the loop again or not. `RESPONSE(1..3)` contained unspecified characters, and `LENGTH` had the value 0, so it wasn't equal to YES. `DONE` remained FALSE and it went to the top of the loop again. The solution to the problem is to add a `Skip_Line` immediately after `Get(Y)`. This gets past the line terminator so `Get_Line` will wait for you to enter a response.

Notice I have called `New_Line` following each `Get`. That's because I expect `Get` to recognize the carriage return and throw it away without echoing it. I tried `TEXT_IO_Quirk` on two different Ada compilers. On one, that's exactly what happened. The prompt to input Y was on the line immediately below the input X prompt because it did not echo the `<CR><LF>` sequence. On the other, there was a blank line between the two prompts because it did echo the `<CR><LF>`. The spec doesn't say which is correct, so both are correct. If I had left the `New_Line` calls out, then one machine would display prompts on adjacent lines, but the other machine would overlap the prompts.

When porting a program from one computer to another, you may find that you have to add (or delete) `New_Line` calls after every `Get` and `Get_Line`. If you want to `Put`, `Get`, `Put`, and `Get`, all on the same line, some implementations of `TEXT_IO` will prevent you from doing that because the carriage return that terminates the first `Get` will be automatically echoed.

Both machines acted the same when I entered a blank line in response to the input X prompt. They ignored the carriage return internally, and continued to wait for me to enter X without raising an exception. What surprised me was that both echoed the `<CR><LF>` sequence.

Other User Interfaces

I hope you agree that Ad Hoc solutions like the one shown above aren't a very good idea. Every time you port a program, you'll have to tweek on it to

make it work. There's got to be a better way. I think the better way is to not try to use `TEXT_IO` as a user interface. It wasn't designed to be a user interface, doesn't have enough capability, and it isn't consistently implemented.

Remember, Ada doesn't require you to use `TEXT_IO`. `TEXT_IO` is just another feature that you can choose to use if you like. I don't like. Instead, I wrote my own set of user interfaces. These packages are called `VIRTUAL_TERMINAL`, `SCROLL_TERMINAL`, and `FORM_TERMINAL`. A complete description of these user interface packages (with source code) is in Ada in Action.²

VIRTUAL_TERMINAL

Terminals are notoriously inconsistent when it comes to control codes. They all have different control sequences for clearing the screen and moving the cursor. The `VIRTUAL_TERMINAL` hides all these differences. It can be used for screen-oriented displays. It is handy whenever you want to move the cursor all over the screen and write text fragments in different places, but that isn't its main use. The `VIRTUAL_TERMINAL` is most valuable as a foundation for other terminal packages, such as `SCROLL_TERMINAL` and `FORM_TERMINAL`. Those two packages are built entirely on top of `VIRTUAL_TERMINAL`, with no system-dependent calls, so it isn't necessary to have different bodies for every implementation. If you can write a `VIRTUAL_TERMINAL` body that works with a different physical terminal or different operating system, then you can port `SCROLL_TERMINAL` and `FORM_TERMINAL` without any modifications.

SCROLL_TERMINAL

`SCROLL_TERMINAL` is `TEXT_IO` redesigned for users instead of files. It contains familiar subprograms like `Get`, `Get_Line`, `Put`, `Put_Line`, `Set_Col`, and so on. The difference between it and `TEXT_IO` is that it supports line editing consistently, offers defaults, allows the user to enter invisible data, and has built-in `NEEDS_HELP` and `PANIC` exceptions. It never echoes the terminating carriage return, regardless of the host operating system, so you can confidently follow every `Get` with a `New_Line` if you want the

next prompt to appear on the next line. (You can leave out the `New_Line` if you want the next prompt to appear on the same line.) It is good for user dialogs, where questions must be asked and answered in a specific order.

FORM_TERMINAL

`FORM_TERMINAL` is radically different. It has the same editing features and `NEEDS_HELP` and `PANIC` exceptions that `SCROLL_TERMINAL` has, but the similarity stops there. `FORM_TERMINAL` fills the screen with questions, default responses, and spaces for user inputs. The user can jump around the screen, entering data in any order. The user can even go back to previous screens if necessary. You could write a spreadsheet program using the `FORM_TERMINAL`. (Try doing that with `TEXT_IO`.)

Other File Interfaces

I've had only minor trouble with `TEXT_IO` as a file interface. Sooner or later, though, I'm afraid `TEXT_IO` could cause some major problems. Just to be on the safe side, I'm working on a package called `ASCII_IO` that is a portable version of `TEXT_IO`. It has the same features as `TEXT_IO`, but it operates exactly the same on all operating systems. Initial experiments with `ASCII_IO` show that it solves the problems I've talked about here, but it creates a whole new set of problems. See the March/April '89 Ada Info column³ for a discussion of those problems.

Tips For Using TEXT IO

I use `TEXT_IO` in simple example programs when I don't want irrelevant IO questions to distract from the point of the example. For real programs, however, I never use `TEXT_IO` for a user interface, and I'm working on a better text file interface. My first tip is:

(1) Don't use TEXT IO if you can avoid it.

If you take that advice, you don't need any more tips. However, if you are stuck with `TEXT_IO`, here are some more helpful suggestions.

(2) Limit your input routines to Get_Line (and possibly Skip_Line).

You saw the trouble you can get into when you mix *Get* and *Get_Line* in Figure 1. You avoid this if you always use *Get_Line* to read characters into a text string that is longer than the longest possible input string. This limits you to one value on a line, which costs a little overhead (extra <CR><LF> sequences), but if you were worried about file size you would be using binary files instead of ASCII files. I like one value per line because it makes it easy to examine the file (characters don't fall off the side of a printed listing) and it is easy to find the value of a specific variable (the Nth variable is on the Nth line).

(3) Use Meaningful terms or abbreviations for boolean variables.

Suppose you have a variable called *PROTECTED* that can *TRUE* or *FALSE*, and you need to store this variable in a file. People commonly instantiate *ENUMERATION_IO*. I don't like that solution. It forces you to use *Get* instead of *Get_Line*, because *ENUMERATION_IO* doesn't have a *Get_Line*. Here's an alternative:

```
-- save a boolean variable in a file
put_line
  (FILE,boolean'IMAGE(PROTECTED));

-- read a boolean variable back from
-- a file (TEXT'LENGTH > 5)
get_line(FILE, TEXT, LENGTH);
PROTECTED :=
  boolean'VALUE(TEXT(1..LENGTH));
-- raises CONSTRAINT_ERROR for
-- strings other than TRUE or FALSE
```

Suppose you print a file containing many boolean variables. It will be full of the words *TRUE* and *FALSE*. It may not be clear which variables are *TRUE* and which are *FALSE* simply by looking at the file. That's why I prefer to do it this way:

```
-- save a boolean variable
if PROTECTED then
  put_line(FILE,"PROTECTED");
else
  put_line(FILE,"UNPROTECTED");
end if;

-- read it back (TEXT'LENGTH > 11)
get_line(FILE,TEXT, LENGTH);
if TEXT(1..3) = "PRO" then
  PROTECTED := TRUE;
elsif TEXT(1..3) = "UNP" then
  PROTECTED := FALSE;
else
  raise CONSTRAINT_ERROR;
end if;
```

(4) Use IMAGE and VALUE attributes for Integer_IO.

You don't to have instantiate *INTEGER_IO* to input or output integers. You can use the *IMAGE* and *VALUE* attributes to read and write corresponding text strings.

```
-- save an integer variable in a file
put_line (FILE,integer'IMAGE(X));

-- read an integer variable back from
-- a file
get_line(FILE, TEXT, LENGTH);
X := integer'VALUE(TEXT(1..LENGTH));
-- may raise CONSTRAINT_ERROR
```

(5) Use Image and Value functions for Real number IO.

Ada doesn't have *IMAGE* and *VALUE* attributes for variables of type *float*, but you can write *Image* and *Value* functions that do conversions between real numbers and character strings. I've done that already myself. The source code for those functions is in the *ASCII_UTILITIES* package in *Ada in Action*.¹

Conclusion

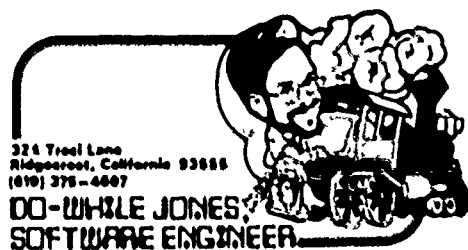
TEXT_IO tries to be both a file interface and a user interface, and it does neither job very well. It is loosely specified, presumably to make it compatible with a variety of underlying operating systems, and this leads to portability problems. It is adequate for trivial programs, but just won't do the job for programs with an extensive user interface, or programs that have to share text files on a variety of different machines.

One solution would be to make massive changes to Chapter 14 of the LRM. That's not a good idea because the revision of MIL-STD-1815A will take a long time, and significant changes to Chapter 14 will just delay the approval even more. Some vendors will cry "foul" because it will be more difficult to implement the new `TEXT_IO` on their operating systems than their competitors, and they will try to prevent approval.

Fortunately, isn't necessary to change the LRM. It doesn't say you have to use `TEXT_IO` for all IO. You can simply ignore `TEXT_IO` and use something else for the user interface. If `TEXT_IO` causes compatibility problems, use a different file interface package. You can write your own IO packages, or use packages published in the open literature.

References

1. VAX and VMS are trademarks of Digital Equipment Corporation.
2. *Ada in Action* by Do-While Jones. Published by John Wiley & Sons, Inc. 605 Third Ave. New York, NY 10158
3. "Ada Info" by Do-While Jones. *Journal of Pascal, Ada, & Modula-2*. Vol. 8 No. 2. March/April 1989. Published by JPAM, Inc. P.O. Box 6338, Woodland Park, CO 80866



AUTHOR'S BIOGRAPHY

Do-While Jones writes the Ada Info column for the *Journal of Pascal, Ada, and Modula-2*. He has also published articles on Ada in *Dr. Dobbs' Journal of Software Tools and Computer Language* magazine.

In 1971, Do-While Jones received the degree of Bachelor of Science (with distinction) in Electrical Engineering, from a midwestern university better known for its football team than its engineering school. Since graduation he has been employed in the defense industry of a well-known free-world nation. During the course of that employment he was granted a patent for a radar signal processing algorithm.

Mr. Jones began programming in Ada before most Ada compilers on the market today were commercially available. He was a beta test site for two Ada compilers, and has evaluated six others.

Automatic Test Data Generation and Assertion Testing for Ada Program Units

Lauren Mayes, Rhonda Wienk Aragon, Deborah Terrien, Julie Trost

Intermetrics Inc., Huntington Beach California

Under the auspices of the Software Technology for Adaptable, Reliable Systems (STARS) Foundations program, Intermetrics Inc. developed a tool to support the testing of Ada program units. The tool, called the Ada Test Support Tool (TST), is a compiler independent, portable tool used for testing subprograms and task entry points within compilable Ada units. TST supports the automated testing of Ada program units, allows assertions to be made about test results, documents test results, and provides for regression testing. This paper describes TST and experiences gained in the development and use of the tool.

INTRODUCTION

As the demand for highly reliable software systems grows, software testing methodologies and tools become increasingly important. New testing methodologies, like those described in [GelHis], which focus on preventative software testing throughout the life-cycle show promise in meeting these demands. The Ada Test Support Tool (TST) developed by Intermetrics can be used to automate some of the activities required in a life-cycle testing approach.

TST is a dynamic analysis, compiler independent tool written in Ada to test Ada subprograms and tasks, collectively termed routines for this paper. TST generates Control Programs that contain calls to visible routines in Ada units. Users invoke the Control Program and supply input parameters or request "test data generation" for routines they choose to test. Assertions may be made about output values to specify the expected result of tests. Input parameters and test results are output to a TST report.

This paper describes TST and the lessons we learned while developing the tool. Topics presented include: Ada as a development language, experiences in reusing software, most useful application of TST, problems with the tool, and future directions.

BACKGROUND

TST leverages on technology developed for the Ada Test and Analysis Tools (ATEST) Intermetrics built for the WIS (WWMCCS Information Systems) program. The ATEST tools, documented in [Inter], include a performance analyzer, path analyzer, variable trace tool and a symbolic debugger; all of the tools use dynamic analysis techniques to monitor programs as they are executing. The performance analyzer measures execution speed and the path analyzer records the statements and subprograms executed during the run of a

program. The variable trace tool records the values of program variables during execution of a program and the symbolic debugger allows programmers to step through a program's execution and change the value of program variables at the source code level.

The unique aspect of the ATEST tools is that they are not dependent on a specific Ada compilation system. The system independent nature of the tools is accomplished using a Source Instrumenter that parses Ada programs and embeds additional code in the source code. This code provides "hooks" to a Run Time Monitor that is used to record program execution information. TST uses a modified version of the symbolic debugger's Source Instrumenter.

In 1987, the STARS (Software Technology for Adaptable Reliable Systems) Foundations program contracted a variety of tools targeted to Ada software development, with the intention of promoting a "software first" software development strategy described in [STARS]. "Software first" refers to developing software incrementally and deferring hardware choices to later phases in the development process. The Foundations tools were required to be highly portable for easy integration into software development environments that are being built under the STARS Competing Primes program. Intermetrics proposed TST in two phases. In Phase I, a basic testing capability was provided, and for Phase II, automatic test data generation and assertions were added.

PREPARING A UNIT FOR TESTING

Figure 1 illustrates how users generate TST Control Programs. At the top of the figure, a computer terminal shows the commands which the user enters. The three-dimensional

boxes in the figure represent executable programs. The Shell and Source Instrumenter are provided by TST, the Compiler and Linker are provided by the user, and the Control Program is generated by TST. Each of these programs may be separately executed at the system level with Ada procedure calls or from within the Shell. The Shell provides a help facility, prompts for parameters when programs are invoked, and allows users to set TST system variables (e.g., report width and length, screen echo flag).

First, the user invokes the Source Instrumenter which generates a Control Program and inserts "hooks" into the source code so that the Run Time Monitor can gain control during program execution. The unit to be tested needs to be instrumented along with any units declaring types that are used by the unit being tested. For example, in Figure 2 the procedure `UNIQUE_FILENAME` in package `FILENAME` has a parameter of type `SYSTEM_DEPENDENCIES.FILENAME`. In order to test the `FILENAME` package, both the `SYSTEM_DEPENDENCIES` and `FILENAME` packages must be instrumented.

The Source Instrumenter adds code to the unit's body for tracing statements but makes no changes to the specification. A support package which provides routines to read, write, get the next value, and compare values is generated for each visible Ada type declaration. In addition, a Control Program which has calls to all routines visible in the unit to be tested is generated. The support package, the Control Program, and the body of the unit being instrumented are copied into a file which is named by appending the extension ".INS" to the body file name.

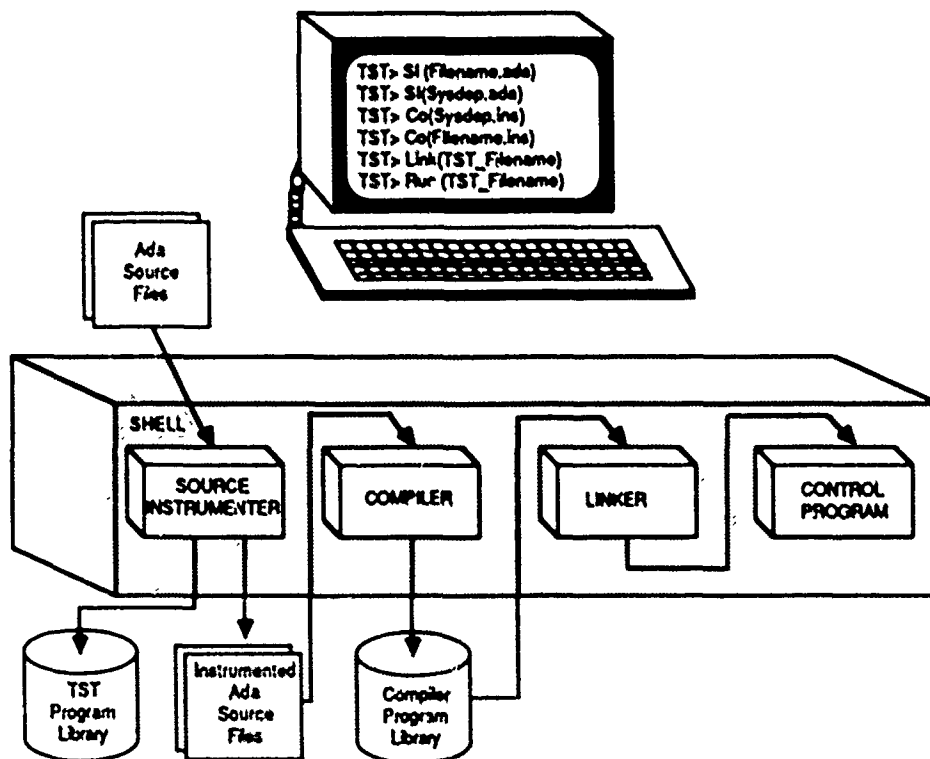


Figure 1. Creating a Control Program.

```

with SYSTEM_DEPENDENCIES;
package FILENAME is
  procedure UNIQUE_FILENAME (
    NAME : out SYSTEM_DEPENDENCIES.FILENAME);
  .
  .
end FILENAME;

```

Figure 2. Both the Tested Unit and Dependent Unit Need To Be Instrumented.

After all units have been instrumented, the compiler is invoked. Any compiler may be used, as long as the TST installation process has been completed for that compiler. Intermetrics has hosted TST on the Alslys PC/AT, Alslys Sun, and the DEC Ada compilers.

When all units have been compiled, the linker is invoked to produce an executable Control Program. The main program input to the linker is the Control Program generated during instrumentation. The Support Program, Control Program, specification of the unit being tested, any dependent units, and the units comprising the Run Time Monitor are linked to create an executable Control Program. The Run Time Monitor units must be compiled prior to linking; this is done once when TST is installed and does not have to be repeated each time an executable Control Program is created.

TESTING A UNIT

After the executable Control Program is created, testing may begin. When the Control Program is executed from the Shell or at the system level, a Testing Subsystem user interface is displayed. A TST system variable specifies whether or not the Testing Subsystem is in full screen or line mode. For line mode, only the Testing Subsystem prompt, "T>" is displayed. If the terminal supports full screen mode, then a dual window Testing Subsystem is displayed. The upper window lists testable routines and assertions, and the lower window is used for interaction between TST and the user. Figure 3 illustrates the full screen Testing Subsystem for an Exchange package which contains two procedures for swapping integer and string values.

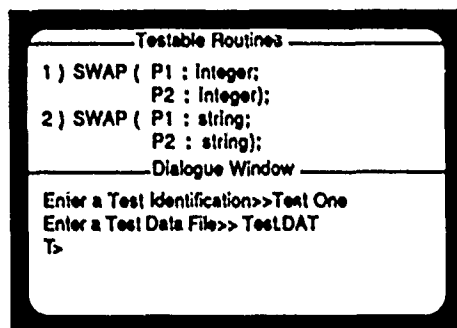


Figure 3. The Testing Subsystem Displays Testable Routines and Prompt for Input

While in the Testing Subsystem, users may call routines, make assertions, generate test data and invoke the help facility by using Testing Subsystem commands. Table 1 lists and describes each command.

As shown in Figure 3, the user is prompted for some information about the test. The user is first prompted for a textual description of the

Table 1. Commands Are Used To Get User Input.

COMMAND	DESCRIPTION
ASSERTION_HANDLING (ACTION)	Set assertion failure action
CALL_ROUTINE (ROUTINE_ID)	Call a routine to test
DELETE_GLOBAL (ASSERTION_ID)	Delete a global assertion
DELETE_LOCAL (ASSERTION_ID)	Delete a local assertion
DISPLAY_ASSERTIONS	List current assertions
DOWN (NUM_LINES)	Scroll down NUM_LINES
END	Quit the Testing Subsystem
GLOBAL_ASSERT (ASSERTION)	Make a global assertion
HELP(TOPIC)	Get help for a command
LIST_ROUTINES	List routines in unit
LOCAL_ASSERT (ASSERTION)	Make a local assertion
UP (NUM_LINES)	Scroll up NUM_LINES
WINDOW (DISPLAY_WINDOW_SIZE)	Resize upper window

test session. This description may be used to identify and track tests. The Test Data File is the name of an ASCII text file that will be used to record commands input by the user during the testing session. If the Test Data File entered already exists, then the user will be prompted to either use that file as test input or to overwrite the file with new commands. The Test Data File provides a convenient means for repeating tests (regression testing).

If a Test Data File was specified as input, then the commands in the file will be run without user interaction. Otherwise the "T>" prompt is displayed and the user may start entering commands. The commands of the most significance are CALL_ROUTINE, GLOBAL_ASSERT, and LOCAL_ASSERT. Each of these commands is described below. The Exchange package shown in Figure 3 is used as an example for the descriptions. In the examples, TST prompts are indented and user input is in boldface type.

The CALL_ROUTINE command allows the user to specify test cases for a routine. For example, if the user issues the command:

T> CALL_ROUTINE 1

for the Exchange package, then the system will echo the name of the routine and wait for the user to enter parameter values:

T> CALL_ROUTINE 1

SWAP (

The user then enters literal values for the test and test results are echoed to the screen:

T> CALL_ROUTINE 1

SWAP (4, 6);

P1 = 6

P2 = 4

T>

or generates test data:

T>CALL_ROUTINE 1

SWAP (5, *1);

.....

Input Data:

P1 = 5

P2 = -32768

P1 = -32768

P2 = 5

.....

Input Data:

P1 = 5

P2 = 0

P1 = 0

P2 = 5

.....

Input Data:

P1 = 5

P2 = 32767

P1 = 32767

P2 = 5

T>

Test data generation is accomplished through the use of the '*' symbol. If '*' is entered, then every possible value for that type will be generated. If '*X' is entered, where X is a natural number, then the set of possible values for the type will be partitioned into X subsets, and the first, middle, and last values for each of the subsets will be generated. All permutations of parameter values will be generated. Figure 4 illustrates the permutations generated for an enumerated type.

Another feature available in TST is assertions. By using the GLOBAL_ASSERT and LOCAL_ASSERT commands, the user can specify conditional statements about output parameters and function results. For example, for the Exchange package, the following assertions could be made:

T>GLOBAL_ASSERT 1, P1 <10

Global Assertion 1) 1, P1 <10

T>LOCAL_ASSERT 1, P2 >5

Local Assertion 2) 1, P2 >5

where the first number in the command represents the routine to which the assertion applies and the following input is a conditional statement about an output value. In this case we have made a global assertion about the integer Swap procedure that states that the value of parameter P1 should be less than 10. The local assertion states that for the integer Swap procedure, parameter P2 should be greater than 5. The example below shows how assertions fail.

T>CALL_ROUTINE 1

SWAP(5, 3);

P1 = 3

P2 = 5

*** Local Assert 2) 1, P2 > 5 Failed

T>CALL_ROUTINE 1

SWAP(6, 11);

P1 = 11

P2 = 6

*** Global Assert 1) 1, P1 <10 Failed

T>

In the first call the local assertion failed because the value of P2 was less than 5. In the second call the global assertion failed because the value of P1 was greater than 10. The local

assertion did not fail for the second call because local assertions are valid only for the next CALL_ROUTINE command. Global assertions are valid until the testing session ends or the assertion is deleted using the DELETE_GLOBAL command. Current assertions may be displayed in the upper window of the Testing Subsystem using the DISPLAY_ASSERTIONS command.

If A Package Has The Declarations:

type COLOR is (RED, BLUE, GREEN, YELLOW, BROWN);
procedure SWAP (C1 : in out COLOR, C2 : in out COLOR);

And The Following Command Is Invoked In The
Testing Subsystem
T> C 1

SWAP (*, *1)

Then Swap Will Automatically Be Called With Each Of The
Following Input Values:

SWAP (RED, RED)	SWAP (YELLOW, RED)
SWAP (RED, GREEN)	SWAP (YELLOW, GREEN)
SWAP (RED, BROWN)	SWAP (YELLOW, BROWN)
SWAP (BLUE, RED)	SWAP (BROWN, RED)
SWAP (BLUE, GREEN)	SWAP (BROWN, GREEN)
SWAP (BLUE, BROWN)	SWAP (BROWN, BROWN)
SWAP (GREEN, RED)	
SWAP (GREEN, GREEN)	
SWAP (GREEN, BROWN)	

Figure 4. All Permutations of Parameter Values Are Generated.

Assertions provide the user with a mechanism for stating the expected values of output data. When the expected results are not attained, the user is warned by a failed assertion. This is an important concept in that the user is explicitly warned and the warning is put into a report for later review. Failed assertions indicate that a test did not proceed successfully. Assertions are especially valuable for creating test cases using Test Data Files in a text editor. Test Data Files that include assertions about expected test results can be written early on in the software development life-cycle and are useful for discovering problems in software designs.

DOCUMENTATION OF TEST RESULTS

After the user has completed a test session, a collection of reports describing the results is generated. The Configuration Report lists information about the time, date, executable

Control Program name, and default TST system parameters. The Routine Report lists the testable routines that were displayed in the upper window of the Testing Subsystem. The Parameter Report lists the routines that were called, results of tests, assertions made, and failed assertions. Figure 5 shows a Parameter

```

=====
Unit Under Test: ( 1)
procedure SWAP(
  P1 : in out INTEGER;
  P2 : in out INTEGER);

```

Parameter	Entering Value	Exiting Value
P1	4	6
P2	6	4

```

=====
Unit Under Test: ( 1)
procedure SWAP(
  P1 : in out INTEGER;
  P2 : in out INTEGER);
Test Data Automatically Generated
P2 => -1

```

Parameter	Entering Value	Exiting Value
P1	5	-32768
P2	-32768	5
P1	5	0
P2	0	5
P1	5	32767
P2	32767	5

```

=====
GLOBAL Assertion 1) 1, p1 < 10
=====
LOCAL Assertion 2) 1, p2 > 5
Unit Under Test: ( 1)
procedure SWAP(
  P1 : in out INTEGER;
  P2 : in out INTEGER);

```

Parameter	Entering Value	Exiting Value
P1	5	3
P2	3	5

```

*** LOCAL Assertion 2) 1, P2 > 5 Failed
=====
LOCAL Assertion 2) 1, P2 > 5 Deleted
Unit Under Test: ( 1)
procedure SWAP(
  P1 : in out INTEGER;
  P2 : in out INTEGER);

```

Parameter	Entering Value	Exiting Value
P1	6	11
P2	11	6

```

*** GLOBAL Assertion 1) 1, P1 < 10 Failed

```

Figure 5. The Parameter Report Lists Test Results.

Report corresponding to the example given in the text above.

The Execution History Report consists of two parts. The first report lists the order that routines were entered, resumed, or ended. Exceptions are also shown in this report if they were not handled by the originating routine. The second report enumerates the number of times that statements or groups of statements were executed. A listing file that maps statements to the numbers shown in the Execution History Report is produced by the Source Instrumenter. Figure 6 shows an example Execution History Report for the example above.

```

Begin EXCHANGE.SWAP
  (1-4)
End EXCHANGE.SWAP
Begin EXCHANGE.SWAP
  (1-4)
End EXCHANGE.SWAP
Begin EXCHANGE.SWAP
  (1-4)
End EXCHANGE.SWAP
Begin EXCHANGE.SWAP
  (1-4)
End EXCHANGE.SWAP
Begin EXCHANGE.SWAP
  (1-4)
End EXCHANGE.SWAP
Begin EXCHANGE.SWAP
  (1-4)
End EXCHANGE.SWAP
Begin EXCHANGE.SWAP
  (1-4)
End EXCHANGE.SWAP

```

Statement	Execution Count
EXCHANGE	
(1-4)	6
(5-8)	0

Figure 6. The Execution History Report Shows Routines and Statements Executed.

The Execution History Report is useful for ensuring that all routines and statements were executed.

The Test Data File generated while the test was proceeding may also be examined for determining test coverage. The Test Data File is in ASCII format, includes comments, and may be modified or created by the user. The Test Data File created in the test session above is shown in Figure 7.

```

-- C 1
-- 4,6)
CALL ROUTINE 1 (
P1 => 4
P2 => 6
)
-- C 1
-- 5, 11)
CALL ROUTINE 1 (
P1 => 5
P2 => -32768
)
CALL ROUTINE 1 (
P1 => 5
P2 => 0
)
CALL ROUTINE 1 (
P1 => 5
P2 => 32767
)
ASSERTION HANDLING continue
GLOBAL ASSERTION 1, P1 < 10
LOCAL ASSERTION 1, P2 > 5
-- C 1
-- 5,3)
CALL ROUTINE 1 (
P1 => 5
P2 => 3
)
-- C 1
-- 6,11)
CALL ROUTINE 1 (
P1 => 6
P2 => 11
)

```

Figure 7. The Test Data File May Be Viewed And Modified.

RECOMMENDED APPLICATIONS OF TST

Using test data generation and assertion handling features, we have found that TST is a good tool for specifying tests when designing and during the final stages of coding an Ada unit. Some recommendations about using TST are listed below:

- Routines that are encapsulated and have well-defined inputs and outputs are the best candidates for testing with TST.
- One of the most promising application areas for TST is in testing reusable software components that have well-specified outputs. As software reuse becomes more widely accepted and practiced, it will become increasingly important to prove the correctness of the reusable software. Conversely, reuse is promoted when programmer confidence is increased because it is easier to test and greater test coverage is ensured. TST allows programmers to easily create new test cases for software on the compiler that will be used for development and delivery, thus improving the confidence in reusable software that is ported to different machines and compilation systems.
- TST could be used as a debugging tool, but this is not recommended because compile times are increased because of the support package and Control Programs that the Source Instrumenter generates.
- Because TST focuses on the inputs and outputs of routines, those routines that modify global data and do not return values cannot be adequately tested with TST.
- Components that are highly user interactive, like menu generators and screen generators could be tested quite easily when the user is present to view the output on the display device.
- To provide thorough testing, TST should be used in the context of an integrated test plan.

PROBLEMS WITH TST (AND POSSIBLE SOLUTIONS)

We have found that a significant amount of code is generated during instrumentation for test data generation. During instrumentation, a support package that includes routines for reading, writing, getting the next value, and doing comparisons for assertions are generated for each visible type in a package specification. Table 2 lists the lines of code generated for the different types. Packages containing many types, especially complex user-defined types, could result in very large support packages.

Table 2. Code Added When Instrumenting.

CODE GENERATED TYPES	Assertions		Test Data Generation		Read/Write	
	stm	lines	stm	lines	stm	lines
Predefined	0	0	0	0	0	0
Enumeration, Floating & Fixed Point	1	1	2	10	2	2
Access	35	69	18	39	28	56
Array constrained	38	72	88	162	28	59
unconstrained			122	227	36	76
Record no discriminant	39	72	98	180	25	55
discriminated			155	300	40	78

stm: Ada statements

lines: lines of code (includes comments and blank spaces)

In addition, Control Programs are generated for each unit and a small amount of code is added at the beginning and end of each routine. The size of Control Programs is dependent on the number of routines defined in the unit being tested and the number and type of parameters within each routine.

A significant amount of the generated code may never be used. For example, package A

may use one type declared in package B which has ten additional types defined. Package B must be instrumented in order to test package A and code will be generated for all of the types in package B because when B is instrumented we do not know which types A will need.

While a good Ada linker will remove this "dead" code, the problem of compiling the additional code is not alleviated. A possible solution to this problem would be to provide a post-instrument tool that allows the user to specify which unit they plan to test. The tool could go through the source instrumented code for package B and comment out the code added for the types that are not needed along with the Control Program, if any, created for B.

Another problem is the technique used for test data generation. Very large test inputs will result for routines having many parameters which have a large range of possible values. Using the partition method of test data generation (i.e., *X) alleviates this problem somewhat, but does not allow complete test coverage. We have considered adding the ability to generate values within a specified range, but even this technique may produce more data than is needed. These problems point out the fact that at the present time, tools cannot bear the full burden of testing; intelligent selection of test input must be provided.

Currently TST has no facility to do configuration control on the testing of many units. A test manager tool could be provided to inform the user when a unit has been changed and needs to be re-tested. A tool that lists the units that need to be instrumented to test a particular unit would also be beneficial.

Additional reports could be generated that show thoroughness of testing for each routine. A tool that allows testing of routines within the body of units would also be useful. This could be accomplished by putting the Control Program in the body of the package being tested and adding code for all types declared in the body.

ADA AS A DEVELOPMENT LANGUAGE

Similar tools [Deutsch] have been created for testing programs written in other languages. One of the problems we encountered developing the tool in Ada was the inability to get "into" the code because of scoping rules and strong typing. For example, private and limited private types cannot be tested because Ada does not allow the examination of private data outside of the unit's body. We also encountered problems due to the rich typing provided by Ada. Test data generation for the vast number of types that may be constructed and complex types like multi-dimensional unconstrained arrays and variant records was especially challenging.

Because Ada is well suited to reuse, we were able to build TST in a relatively short period of time. Table 3 shows the amount of code developed and reused and person-months required to compete the tool. User's guides, presentations, and design reviews are included in the person-month estimates.

We had the added advantage of programmers who knew Ada when the project started, and some who had worked on the original WIS tools that formed the basis of TST.

Table 3. Significant Productivity Increases May Be Achieved When Software Is Reused.

CODE DELIVERED	62,500 LOC
REUSED CODE	45,000 LOC
PERSON MONTHS	34.5
PRODUCTIVITY WITH REUSE	1811.5 LOC/Person Month
PRODUCTIVITY NOT INCLUDING REUSE	1304.3 LOC/Person Month

Our experience in reusing software resulted in the following conclusions.

- Software designed using object-oriented techniques is easier to understand and reuse.
- Adequately commented code is important, but not critical in reuse. Code that performs a well-understood function, is well designed, uses meaningful variable names, and is not extremely complex can be understood by experienced Ada programmers even if it is not well documented.
- Isolation of system dependent features makes porting reusable software almost pain-free.

During development of TST, we were pleased with the state of Ada compilers. Most of the development for TST was done on C/AT clones using the Alsys compiler. A few problems concerning data size were encountered, but these were expected. We also had some problems with tasking and our program libraries, but these were corrected with compiler updates.

CONCLUSION

Tools supporting software testing are important in achieving the level of reliability required for today's complex software systems. However, testing tools must be integrated in a complete software test plan that spans the life-cycle. We have outlined a tool that supports testing in a small area, and have given suggestions for improving the tool. It is the opinion of the authors that TST and other test tools will not become widely accepted until industry understands the benefits that testing can bring to both software design and verification.

REFERENCES

- [Deutsch] Deutsch, M.S., *Software Verification and Validation, Realistic Project Approaches*, Prentice-Hall Series in Software Engineering, 1982, p. 130
- [Inter] User's Guide for the Ada Testing And Analysis Tools, Intermetrics, Inc., 1985.
- [GelHis] Gelperin, D., Hetzel, B., The Growth of Software Testing. *Communication of the ACM*, June 1988.
- [STARS] STARS Technical Program Plan, 6 August 1986.

About the Authors:

The authors may be contacted at
(714) 891-4631



Lauren Mayes is a software engineer in the Aerospace Systems Group's Civilian Space Programs Department at Intermetrics. She is the program manager for the Ada Test Support Tool. Her interests include software testing, software reuse, design and development tools, real-time software development, and quality metrics. She received a BS in information and computer science from the University of California at Irvine.



Deborah Terrien is a software engineer in the Aerospace Systems Group's Civilian Space Programs Department at Intermetrics. Her interests include software design and development tool and software reuse. She received a BS in Computer Science from California State University at Fullerton. She is a member of IEEE.



Rhonda Wienk Aragon is a software engineer in the Aerospace Systems Group's Civilian Space Programs Department at Intermetrics. Her interests include software reuse, software design and development tools, and database management and design. She received a BS in computer science from Chapman College. She is a member of IEEE and ACM.



Julie Trost is a software engineer in the Aerospace Systems Group's Civilian Space Programs Department at Intermetrics. Her interests include software design and development tools. She received a BA degree in mathematics with an emphasis in computer science from the California State University at Long Beach.

PRACTICAL ADVICE FOR DESIGNING ADA SYSTEM ARCHITECTURES

Caroline D. Buchman

Allied-Signal Aerospace Company
Computer-Aided Engineering Center
Teterboro, New Jersey 07608

ABSTRACT

Packaging schemes can negatively impact a system's performance and maintainability. Adherence to a design methodology and good software engineering are not always sufficient to ensure optimum system performance. This paper suggests guidelines to be followed to avoid or correct the problems of code copy instantiation, linking, and recompilation. Four areas are covered: generics; packaging; interfaces to non-Ada code; and, recompilation requirements.

Packaging schemes can negatively impact a system's performance and maintainability. Herein are suggested guidelines to minimize this kind of impact. These guidelines are based upon experiences encountered in building two large interactive applications involving many thousands of lines of Ada code.

The design of the software architecture can be as important to the success and acceptance of a system as the design of the logic. More than any other programming language, Ada requires a software engineer to consider the physical arrangement and placement of program units. Program units, data types, and data objects must be grouped into packages.

How packages are organized relates directly to how the finished system will perform and how easily it can be maintained. True, the design methodology must be the primary authority for packaging schemes. Within this framework, however, consideration also must be given to compiler and linker behaviors and Ada recompilation issues.

Idiosyncrasies in compilers and linkers make it difficult to predict how an Ada system will behave when it is finally ported to the target machine. These behavior differences often can be a factor in the system's success. Hence the notion that system design can be completed without regard for the supporting hardware and operating system environment is fundamentally false. Ada designs will be independent of hardware considerations only when compilers and linkers are mature enough to eliminate the code-copy method of instantiation and the philosophy of "link in everything."¹

The present state of the art is such that the approach a compiler/linker takes to generic instantiation or packaged modules can make a critical difference in the deliverable system. Consequently, knowing how the linker and compiler work does and should affect how the Ada software architecture is designed. Moreover, that architecture should take recompilation effects into consideration, particularly in large systems where the recompilation time may be measured in days.

We will focus our attention on four major problem areas, and recommend counter measures that will provide developers with a degree of protection from unpredictable or unacceptable results. The four areas are: 1) generics, 2) packaging, 3) interfaces to non-Ada code, and 4) recompilation requirements.

Compilers that implement each generic instantiation as a separate code copy can cause tremendous overhead in the final image size. Each instantiation of the same package causes a new complete copy of the original generic, with values for its generic parameters, to be generated. The guidelines presented here are intended to minimize the potential code explosion.

Linkers generally link in all of a package whether it is actually used by the application or not. When unnecessary code and data are linked into the final image, memory is wasted; virtual memory systems experience increased page faulting, resulting in performance degradation. The consequences of the link everything philosophy are far reaching, particularly in the area of reusability. We will discuss packaging schemes that are designed to minimize any excess baggage.

The designers of the Ada language realized that the transition from other languages would be gradual and that entire libraries of proven subroutines would have to be accessible from Ada in order for the language to reach acceptance by the engineering community. However, indiscriminate use of Pragma INTERFACE may involve penalties when you try to convert those subroutines to Ada. We will offer some suggestions for keeping those penalties to a minimum.

Ada recompilation requirements, enforced by DoD-STD-1815A, can be costly in large system development; it may take days to recompile a system. Many times these compilations are unnecessary, such as when an additional enumeration value was added to a type definition. Incremental compilation is not widely available. We will demonstrate that a packaging strategy can protect against extensive recompilation.

Generics

Much of the problem associated with generics actually occurs in instantiations. As mentioned before, Ada compilers have implemented instantiation by copying the entire generic unit and substituting the actual generic parameters for the formal generic parameters and then compiling this code. Given this, how can we minimize the size issues raised by creating multiple copies of the generic unit?

Package All Generic Instantiations

Lacking rules to the contrary, package developers will instantiate a generic directly in the package. If two package developers need the same generic, each will instantiate it. If they happen to be using the same parameters, two identical instantiations of the generic are in the system. It is difficult to identify this waste because these instantiations are hidden in package bodies, in accordance with the dictates of encapsulation.

The simplest and cleanest way to handle this problem is to create a package specification for the instantiation and then use rename and subtyping to make the necessary instantiated units and data types visible. Now the generic package is instantiated only once; only one copy of the code is made.

To illustrate this, Example 1 is a commonly used generic specification for a linked list. Example 2 is a package that encapsulates an instantiation of this generic. With this resource available, package developers may make as many uses of the linked list as is necessary without inadvertently creating multiple instantiations.

Include Only the Necessary in a Generic

Generic unit bodies frequently contain supporting program units that do not depend on a generic parameter. There is nothing at all generic about these program units, but they contribute to the functionality of the generic package and hence are included in the package body. Of course, at instantiation, the compiler creates copies of these units as well as those that are truly generic since

they are contained in the same package body. Place these supporting units in their own package and reference the package from the generic unit. The unnecessary code duplication is avoided. This problem is frequently encountered in generic units that are parameterized by generic procedures and/or functions only. All such generic package body routines should be examined to see if they can be moved into a package of their own. Example 3 illustrates such a package.

```
generic
  type LIST_ITEM is private;
package LINKED_LIST is
  type LIST_TYPE_ACCESS is limited private;

  procedure INSERT_AT_HEAD
    (QUEUE : in out LIST_TYPE_ACCESS;
     ITEM   : in   LIST_ITEM);

  procedure INSERT_AT_TAIL
    (QUEUE : in out LIST_TYPE_ACCESS;
     ITEM   : in   LIST_ITEM);

  procedure REMOVE_FROM_HEAD
    (QUEUE : in out LIST_TYPE_ACCESS;
     ITEM   : out  LIST_ITEM);

  procedure REMOVE_FROM_TAIL
    (QUEUE : in out LIST_TYPE_ACCESS;
     ITEM   : out  LIST_ITEM);

private
  type LIST_TYPE;
  type LIST_TYPE_ACCESS is access LIST_TYPE;

  type LIST_TYPE is
    record
      INFO : LIST_ITEM;
      NEXT : LIST_TYPE_ACCESS := null;
      PREV : LIST_TYPE_ACCESS := null;
    end record;
end LINKED_LIST;
```

Example 1
Generic Linked List Package Specification

Packaging

As discussed earlier, linkers link the entire context of a program unit without regard for what is actually necessary to the execution of the program. Many extraneous data objects and program units are included in images this way.

Object Oriented Design (OOD) is a methodology that is frequently associated with Ada, primarily because of Ada's powerful capabilities to encapsulate and abstract data and procedures.² If you subscribe to an OOD methodology, then your package specifications contain all of the object definitions and actions normally associated with the object. Take as an example the management of a

```

with LINKED_LIST;

package ITEM_LIST is

  type ITEM_TYPE is . . . ;

  package ITEM_LINKED_LIST is new LINKED_LIST
    (LIST_ITEM => ITEM_TYPE);

  subtype LIST_TYPE_ACCESS is
    ITEM_LINKED_LIST.LIST_TYPE_ACCESS;

  procedure INSERT_AT_HEAD
    (QUEUE : in out
     ITEM_LINKED_LIST.LIST_TYPE_ACCESS;
     ITEM : in ITEM_TYPE)

  renames ITEM_LINKED_LIST.INSERT_AT_HEAD;

  procedure INSERT_AT_TAIL
    (QUEUE : in out
     ITEM_LINKED_LIST.LIST_TYPE_ACCESS;
     ITEM : in ITEM_TYPE)

  renames ITEM_LINKED_LIST.INSERT_AT_TAIL;

  procedure REMOVE_FROM_HEAD
    (QUEUE : in out
     ITEM_LINKED_LIST.LIST_TYPE_ACCESS;
     ITEM : out ITEM_TYPE)

  renames ITEM_LINKED_LIST.REMOVE_FROM_HEAD;

  procedure REMOVE_FROM_TAIL
    (QUEUE : in out
     ITEM_LINKED_LIST.LIST_TYPE_ACCESS;
     ITEM : out ITEM_TYPE)

  renames ITEM_LINKED_LIST.REMOVE_FROM_TAIL;

end ITEM_LIST;

```

Example 2
Package Specification of an Instantiation of the
Generic Linked List in Example 1. Subtyping and
Renaming Have Been Used for Visibility

dictionary. An object is read from, written to, or deleted from the dictionary. According to OOD, a package (or packages) should exist that specifies the object definition and this set of actions. We will assume for purposes of this example that these are contained in a single package, as in Example 4.

As long as each program using this package needs the read/write/delete functions, the package design offers no problems. But suppose another program in the system only needs to read items from the dictionary. Because the package was designed as a single entity, this second program must carry the code to perform the other functions as well. Not only is there extra code included in the image, but there is the potential that functions other than read are being performed. Referencing a unit makes it potentially callable. From a quality assurance perspective, there is no guarantee that this program does not delete or write to the dictionary in some way.

```

with UNIT_DEFINITION; use UNIT_DEFINITION;

generic
  with procedure PROCESS
    (UNIT : in UNIT_TYPE);
package PROCESS_UNITS is
  procedure PROCESS_UNIT
    (UNIT_TO_PROCESS : in UNIT_TYPE);
  end PROCESS_UNITS;

package body PROCESS_UNITS is

  --This function is not dependent on any generic parameter
  --and is such it can and should be removed to its own
  --supporting package.

  function UNIT_IS_PROCESSABLE
    (UNIT : in UNIT_TYPE) return BOOLEAN is
  begin -- UNIT_IS_PROCESSABLE

  end UNIT_IS_PROCESSABLE;

  procedure PROCESS_UNIT
    (UNIT_TO_PROCESS : in UNIT_TYPE) is

  begin -- PROCESS_UNITS
    if UNIT_IS_PROCESSABLE (UNIT_TO_PROCESS) then
      PROCESS (UNIT => UNIT_TO_PROCESS);
    end if;
  end PROCESS_UNIT;

end PROCESS_UNITS;

```

Example 3
Example of Generic Package Body With non-Generic
Parameter Dependent Program Units

The solution is to break up the original package into smaller units and use the subsystem approach (Example 5), although this does have a tendency to multiply the number of packages in a software system. The example should be implemented as multiple packages: 1) Data Definitions, 2) Read-Only, 3) Write, 4) Delete. (In some cases it might be acceptable to combine the Write and Delete procedures in one package.)

Of course, the original program should be able to view this object and its associated actions as a single entity. So, a fifth package specification is created which references the other four and uses rename and subtyping to transfer visibility (Example 6).

This technique does not work for generic packages, we should note. Using the linked list in Example 1, if a program needed to traverse the list going forward and backward, two more procedures would have to be added to the package specification. This has been done in Example 7.

```

package ITEM_DICTIONARIES is
  type DICTIONARY_ITEM is
    record
      ...
    end record;
  type DICT_ID_TYPE is ...;

  procedure WRITE
    (DICTIONARY_IDENTIFIER : in DICT_ID_TYPE;
     ITEM                  : in DICTIONARY_ITEM);
  procedure READ
    (DICTIONARY_IDENTIFIER : in DICT_ID_TYPE;
     ITEM                  : in out DICTIONARY_ITEM);
  procedure DELETE
    (DICTIONARY_IDENTIFIER : in DICT_ID_TYPE;
     ITEM                  : in DICTIONARY_ITEM);
end ITEM_DICTIONARIES;

```

*Example 4
Package Specification Illustrating Object and Actions*

This change requires all other users of the generic package to recompile, and imposes the necessity to carry around the two traverse routines. There is no elegant solution. Because the unit is generic and generic units cannot instantiate other generic units based upon their own generic parameter values, this package cannot be broken into smaller dependent packages. A tradeoff must be made with regard to final image size and ultimate maintainability. If the effect on image size is critical, the only alternative is to have two packages that manage linked lists, one with the traversing routines and one without.

Structured Design Practices Support Architectural Changes

We have found many times that adhering to principles of good structured design can help to overcome problems in the architecture. In the following actual situation, the minimal cohesion between program units in a package that had grown too large made subdividing the package a small chore. Not a single unit body was modified, and the pared program ran through its test suite flawlessly the first time.

We were able to reduce the image of an Ada program by fifty percent by re-packaging some of the units originally written for another program within the same project. Entire trees of unused, but referenced, program units were eliminated by re-packaging. The original size of the program was 1012 blocks (512 bytes/block), but after careful study of the reused code to determine which units were being unnecessarily included in the program closure, and re-packaging to eliminate numerous packages, the image size was reduced to 443 blocks! This also reduced the image load time, and page faults were decreased by thirty-three percent.

```

package ITEM_DICTIONARY_DATA_DEFS is
  type DICTIONARY_ITEM is
    record
      ...
    end record;

  type DICT_ID_TYPE is ...;
end ITEM_DICTIONARY_DATA_DEFS;

with ITEM_DICTIONARY_DATA_DEFS;
use ITEM_DICTIONARY_DATA_DEFS;

package ITEM_DICTIONARY_READ is
  procedure READ
    (DICTIONARY_ID : in DICT_ID_TYPE;
     ITEM          : in out DICTIONARY_ITEM);
end ITEM_DICTIONARY_READ;

with ITEM_DICTIONARY_DATA_DEFS;
use ITEM_DICTIONARY_DATA_DEFS;

package ITEM_DICTIONARY_WRITE is
  procedure WRITE
    (DICTIONARY_ID : in DICT_ID_TYPE;
     ITEM          : in DICTIONARY_ITEM);
end ITEM_DICTIONARY_WRITE;

with ITEM_DICTIONARY_DATA_DEFS;
use ITEM_DICTIONARY_DATA_DEFS;

package ITEM_DICTIONARY_DELETE is
  procedure DELETE
    (DICTIONARY_ID : in DICT_ID_TYPE;
     ITEM          : in DICTIONARY_ITEM);
end ITEM_DICTIONARY_DELETE;

```

*Example 5
Package of Example 4 Subdivided Into Smaller More
Reusable Components*

Interfaces to Non-Ada Code

Existing libraries of non-Ada code are often useful in Ada applications. Some Ada applications may require a portion of the system to be implemented in another language (e.g., Assembler) in order to meet requirements of timing. For these and other reasons, it is often necessary for Ada programs to coexist with other source language solutions.

In order to refer to non-Ada modules from an Ada program, Ada requires an interface definition, including a "pragma INTERFACE." These can be expressed wherever they are needed, but the specter of maintenance difficulties is introduced.

```

with ITEM_DICTIONARY_DATA_DEFS;
with ITEM_DICTIONARY_READ;
with ITEM_DICTIONARY_WRITE;
with ITEM_DICTIONARY_DELETE;

package ITEM_DICTIONARY is
  subtype DICTIONARY_ITEM is
    ITEM_DICTIONARY_DATA_DEFS.DICTIONARY_ITEM;

  procedure READ
    (DICTIONARY_ID : in DICTIONARY_ITEM;
     ITEM : in out DICTIONARY_ITEM)
    renames ITEM_DICTIONARY_READ.READ;

  procedure WRITE
    (DICTIONARY_ID : in DICTIONARY_ITEM;
     ITEM : in out DICTIONARY_ITEM)
    renames ITEM_DICTIONARY_WRITE.WRITE;

  procedure DELETE
    (DICTIONARY_ID : in DICTIONARY_ITEM;
     ITEM : in out DICTIONARY_ITEM)
    renames ITEM_DICTIONARY_DELETE.DELETE;

end ITEM_DICTIONARY;

```

*Example 6
Single Package Specification Created
From the Packages in Example 5*

How, for example, can these definitions be located when one of these modules is to be rewritten in Ada?

Isolate the Interfaces

Treat all non-Ada modules as "foreign" package bodies by defining their own package specifications. By putting the interface definitions in their own specification, the problem of multiple definitions for a single routine is avoided. Since there is a "single source" for the routine, maintenance is simplified.

If you do not package the specifications for a non-Ada interface, then in order to implement that module in Ada, you must strip out the interface definitions, and replace them with the new routine by modifying the original package and, perhaps, producing an additional one. By defining the interface as a package specification, you will simply remove the "pragma INTERFACE" from the package specification, add a package body, and recompile. No other unit need be modified.

(Incidentally, here is an example of unnecessary recompilation being required by Ada. Removal of the pragma INTERFACE should not cause obsolescence of units referencing this package specification.)

```

generic
  type LIST_ITEM is private;

package LINKED_LIST_WITH_TRAVERSE is
  type LIST_TYPE_ACCESS is limited private;
  procedure INSERT_AT_HEAD
    (QUEUE : in out LIST_TYPE_ACCESS;
     ITEM : in LIST_ITEM);

  procedure INSERT_AT_TAIL
    (QUEUE : in out LIST_TYPE_ACCESS;
     ITEM : in LIST_ITEM);

  procedure REMOVE_FROM_HEAD
    (QUEUE : in out LIST_TYPE_ACCESS;
     ITEM : out LIST_ITEM);

  procedure REMOVE_FROM_TAIL
    (QUEUE : in out LIST_TYPE_ACCESS;
     ITEM : out LIST_ITEM);

  procedure TRAVERSE_FORWARD
    (QUEUE : in LIST_TYPE_ACCESS;
     ITEM : out LIST_ITEM;
     CONTEXT : in out LIST_TYPE_ACCESS);

  procedure TRAVERSE_BACKWARD
    (QUEUE : in LIST_TYPE_ACCESS;
     ITEM : out LIST_ITEM;
     CONTEXT : in out LIST_TYPE_ACCESS);

private
  type LIST_TYPE;
  type LIST_TYPE_ACCESS is access LIST_TYPE;

  type LIST_TYPE is
    record
      INFO : LIST_ITEM;
      NEXT : LIST_TYPE_ACCESS := null;
      PREV : LIST_TYPE_ACCESS := null;
    end record;

end LINKED_LIST_WITH_TRAVERSE;

```

*Example 7
Linked List With Traverse That Cannot be Subdivided*

Identify What Is Really Needed

Shortly after we formally adopted the Ada language for application development, we were confronted with the need to continue using a large library of approximately 800 FORTRAN subroutines. The prospect of building an interface to 800 routines was daunting. After some research, however, we discovered that our applications only called about 100 of these routines; the rest were low-level supporting program units. The prospect of writing the interfaces to 100 routines was considerably less formidable.

With some linkers, writing an interface to a non-Ada routine is all that is necessary to have the routine included in the final linked image. Under

these circumstances, interfacing to only those routines actually called should be a requirement, not an option.

Organize by Functionality

A package specification of 100 routines is not a manageable architecture. In the case of our FORTRAN library, we broke up the interface into multiple packages, each identified by a specific function; one package for forms, another for cursor movement, another for prompting and reading responses, etc. This not only enhanced our ability to manage the routines, but we also reduced the recompilation necessary when a routine definition had to be corrected or a new definition added.

Ada-ize the Interfaces

Interfaces to operating system calls are frequently necessary, and are a common problem area. In order to be language independent, operating system calls use low-level data typing, i.e., common data typing for many programming languages. A direct translation of these data types weakens the power of Ada's strong data typing.

For example, consider an operating system call that requires a bit mask, one byte in length. Each bit in the mask corresponds to a setting for a particular flag. Of course, this bit mask could be implemented directly as an unsigned byte integer, but then the users of this routine would need to look up the flag that corresponds to each bit and determine the integer value of the flag settings.

By using an Ada record definition, the same bit mask can be defined as a record with each field component corresponding to a bit flag (Example 8). To set the flags, a user need only turn them on or off by name. (Rename can be used if the value needs to be handled as an integer value as well.)

Another common occurrence in operating system calls is limiting an argument to certain values. For example, the formal argument is defined as an integer, the only valid values of which are zero, one, two, and three. A value of zero means scroll up; one, scroll down; two, scroll right; and three, scroll left.

Defining the argument as an INTEGER range 0..3 would be a good start. At least the Ada strong typing and range rules can be used to our advantage, and an out of range value would be signaled through CONSTRAINT_ERROR. But who is going to remember that zero is up? An even "friendlier" type definition would be the enumeration type shown in Example 9.

```
type OS_BIT_MASK is
  record
    FLAG_NAME_FOR_BIT_0 : BOOLEAN := FALSE;
    FLAG_NAME_FOR_BIT_1 : BOOLEAN := FALSE;
    FLAG_NAME_FOR_BIT_2 : BOOLEAN := FALSE;
    FLAG_NAME_FOR_BIT_3 : BOOLEAN := FALSE;
    FLAG_NAME_FOR_BIT_4 : BOOLEAN := FALSE;
    FLAG_NAME_FOR_BIT_5 : BOOLEAN := FALSE;
    FLAG_NAME_FOR_BIT_6 : BOOLEAN := FALSE;
    FLAG_NAME_FOR_BIT_7 : BOOLEAN := FALSE;
  end record;
for OS_BIT_MASK use
  record
    FLAG_NAME_FOR_BIT_0 at 0 range 0 .. 0;
    FLAG_NAME_FOR_BIT_1 at 0 range 1 .. 1;
    FLAG_NAME_FOR_BIT_2 at 0 range 2 .. 2;
    FLAG_NAME_FOR_BIT_3 at 0 range 3 .. 3;
    FLAG_NAME_FOR_BIT_4 at 0 range 4 .. 4;
    FLAG_NAME_FOR_BIT_5 at 0 range 5 .. 5;
    FLAG_NAME_FOR_BIT_6 at 0 range 6 .. 6;
    FLAG_NAME_FOR_BIT_7 at 0 range 7 .. 7;
  end record;
for OS_BIT_MASK'Size use 8;
```

*Example 8
Example of Using Ada's Strong Typing
When Defining Interfaces to Non-Ada Code*

```
type SCROLL_DIRECTIONS is (UP, DOWN, RIGHT, LEFT);
for SCROLL_DIRECTIONS use (UP => 0, DOWN => 1,
  RIGHT => 2, LEFT => 3);
for SCROLL_DIRECTIONS'Size use INTEGER'Size;
```

*Example 9
Example of Using Ada to Specify a More Friendly,
Ada Consistent Interface*

The inherent power of Ada to capture and express the meaning of bit settings and integer values is dramatically illustrated in these two previous examples. We can note that these are capabilities of the language, but we have to elect to make use of them. It takes a certain amount of time and effort to write this code, but the returns in the form of readability and maintainability are manifest.

Recompilation Requirements

This section offers some random suggestions for minimizing the need to recompile and to maximize maintainability. It is an attempt to identify some key principles with the hope that these will spur new ideas and discoveries.

Use Package Naming Conventions

Naming conventions eliminate confusion. For example, give the source code filename the same name as the program unit it contains. This facilitates locating source code files. To differentiate package specifications and bodies, use a trailing underscore on the specification name.

Prefix a program unit's name with the system or subsystem name or acronym. For example, our Ada interfaces to the run-time library are all prefaced with RTL. This makes identifying the run-time library calls and the location of the source code easy, because all RTL packages are stored in one location.

Use Small Packages

Keep the size of Ada packages "manageable." In our experience, this translates into having more smaller packages rather than fewer oversized packages, especially considering the problems with reusability and current linking technology we discussed earlier. Observe the "seven plus-or-minus two" rule of thumb to facilitate comprehensibility. Limit the number of supporting program units in a package body to less than fifteen for the same reason. In general, more smaller packages is more manageable than fewer oversized packages.

Also, try to limit the acceptable nestedness within a particular program unit to four levels. Anything more than four levels of nestedness makes it difficult to understand the program unit logic; the human brain can handle just so much abstraction.

Manage Visibility of Data

Although Ada strongly supports the design principle of encapsulation, it is not unusual to find this principle compromised in Ada programs. One situation in which this commonly occurs is data base access routines. All too frequently the access information is passed from module to module along with the data, even though most of the modules only need to see the data; and a much smaller number do the actual reading and writing.

Separate the data structures from the supporting data objects and program units. This will enable you to limit a referenced unit's visibility to only that information that is absolutely necessary. This provides assurance that only those units that have the responsibility are actually modifying the data base. In a few cases, this technique can reduce recompilation; but it is usually the data structure that gets changed, which means everything becomes obsolete. If another database access routine is added or an existing one modified, recompilation is limited only to those routines that actually access the database.

Place type definitions in their own package specification when they support multiple packages. Each package can then reference the data type definition, subtyping it for visibility if necessary. This advice contains a caveat, however. Be alert for ambiguity resulting from the subtype names when multiple packages, depending on the same common data type, are referenced.

Summary

If you are building large systems with Ada, you are undoubtedly confronted with issues of image size, performance, and maintainability. Some of these issues derive from the way compilers and linkers deal with generics, packages, "foreign" code, and recompilation requirements. This often forces us to be concerned about things that are typically outside the scope of system design.

Compilers and linkers will improve, and designers will not be confronted with these issues within a reasonably near future. In the meantime, these guidelines will enable you to avoid or contend with some of the major architectural concerns.

Acknowledgments

The author wishes to express her appreciation to Mr. Drew Yskamp for his invaluable advice and assistance.

References

1. Firesmith, Donald "Two Impediments to the Proper Use of Ada," Ada Letters September/October, 1987
2. Booch, G., Software Engineering with Ada. Benjamin Cummings Publishing Company. 1983



Caroline D. Buchman
Manager, Mechanical and Software CAE
Allied-Signal Aerospace Company
Computer-Aided Engineering Center, MC 3/12
Route 46
Teterboro, New Jersey 07608

Mrs. Buchman received her Bachelor of Arts in Physics and Mathematics from Smith College in 1978 and her Master of Science in Computer Science from Fairleigh Dickinson University in 1989. She has been involved with Ada since 1983 when Allied-Signal Aerospace Company undertook an effort to develop internal CASE tools specifically targeted to Ada.

ADA DESIGN TOOL

K. Tupper, M. Schwartz, J. Hetzron
S. Barlev, P. Davanzo

Software Technology Department
Shipboard and Ground Systems Group
Unisys Corporation
Great Neck, NY 11020
(516) 574-2337

ABSTRACT

This paper describes the components of the Ada Design Tool (ADT), which is being developed and integrated into a software engineering environment. This environment provides automation and methodology support for life-cycle activities from requirement specification through unit testing. The ADT consists of a graphical and textual editor which enables the software engineer to express an Ada design in Object Oriented Design or Functional Decomposition. In addition to the editors, the ADT has a validation function which ensures that the design is complete and consistent, a source code generator which generates the program templates as well as the detailed code, and a documentation function which produces MIL-STD specifications as well as analytical reports. The ADT will have the capability for specifying and retrieving reusable design templates, as well as prototyping the design.

1.0 Introduction

The software design tools, available to us, supported the traditional development of software which abstracts the top-level design, through the use of a processor model and software architecture descriptions, and the detailed design via coding architecture details and PDL. These design techniques, by and large, are performed in a language-independent manner, thereby requiring the programmer to translate the stated design into the implementation language. Since Ada is more than just a coding language, it is desirable to express our designs in Ada terminology, both at the top level as well as the detailed level, thereby eliminating the need for translation into a target language. Another consideration was that most of the design tools supported a single design method, thereby restricting the designer to the use of a particular methodology and mind-set. Furthermore, the ADT allows the program designer to represent specific features of Ada such as abstraction, multi-tasking, exception handling, encapsulation, and generics.

2.0 Characteristics of the Ada Design Tool

The ADT supports the Software Engineer in the conceptualization, preparation, and generation of Ada programs. The ADT is hosted on a SUN Workstation networking system which gives the user three MIPS of processing power within a multiple-user environment. The workstation uses a multi-window system on a 19-inch, bit-mapped graphical monitor, is menu-driven, and uses a keyboard and mouse interface. As part of a total life-cycle tool set, the ADT diagrams and narratives are stored in the project database to which access is controlled through a relational database management system. This is an ideal environment for high user productivity, programming in the large and small, and access to a centralized repository of Ada designs.

The ADT includes a set of sophisticated editors that simplify the creation and maintenance of several types of design diagrams and textual descriptions. A user may access several diagrams and/or narratives simultaneously through the use of multiple windows. The editors have been designed such that the associations between graphical components and textual counterparts are automatically established. The File Management facilities provide for creating, copying, renaming, deleting, and editing of the database entities. To promote reusability, an import facility is provided, allowing access to elements in other projects.

ADT includes a validation function that ensures all diagrams are syntactically correct and all graphical and textual descriptions are complete and consistent with one another. The ADT will include a facility for generating Ada source code. The source code will be derived by evaluating the graphical views in conjunction with their textual counterparts. The documentation function within the ADT will produce Software Design Documents (SDDs), in compliance with




DOD-STD-2167A, in addition to several analytical reports summarizing design details.


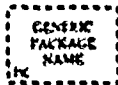
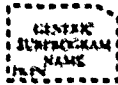
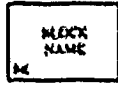


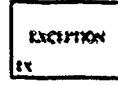
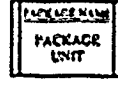
3.0 Ada Design Specification Within ADT

Once the need for a dedicated Ada design facility was recognized, an investigation of several Ada design methods was made. The techniques examined include pictorial and narrative descriptions. While graphical diagrams are beneficial in defining program structures and environmental elements, textual descriptions serve to complete lower-level details and annotations necessary for a complete view. It was concluded that the most effective means of illustrating an Ada design was through a combination of graphical and textual details. Specifically, complete Ada designs can be represented by the series of ADT diagrams (depicted in succeeding pages), textual descriptions, and Ada-compatible PDL.

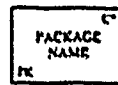
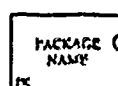
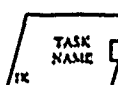

3.1 Graphical Ada Design within ADT. Graphical Ada design within ADT allows the user to represent an Ada design through a series of diagrams composed of Ada-specific icons. In an effort to define a graphical design language to best represent the design of an Ada system and allow for leveled design reviews, an in-depth study of existing graphical techniques was undertaken. As a result of this study, it was decided that the ADT would provide four cooperating graphical views to represent an Ada design. Each view would contain icons that are commonly used and found in many commercially available design tools.

There are three basic classes of icons: Unit icons, Associative icons, and Miscellaneous icons. The Unit icons are used to represent the Ada program units and specific types of programming constructs. Unit icons are connected by Associative icons which represent the relationship (e.g., invocation, signal, access, etc.) between the units. Each icon class has additional annotations to further detail the element or action being represented. The following lists all icons available within the ADT for the Ada-design diagrams.

ICON	NAME	COMMENT
	PACKAGE	Specifies a group of logically related entities, such as types, objects of these types, and subprograms with parameters of these types.
	TASK	Defines concurrent program or code that operates in parallel.
	MULTIPLE OCCURRENCES OF A TASK	Any number of similar tasks of the same type.

ICON	NAME	COMMENT
	SUBPROGRAM	Procedure or function which defines a sequence of code and is invoked by a call.
	GENERIC PACKAGE	A generic package is represented as a standard package using dashed lines to indicate the generic property of the package.
	GENERIC SUBPROGRAM	A generic subprogram is represented as a standard subprogram using dashed lines to indicate the generic property of the procedure or function.
	STANDARD BLOCK	The standard block icon is used to define an Ada block statement which is usually included in an Ada program unit.
	PKL CLUSTER	The PKL cluster enclosures were defined to be code which is usually included in an Ada program unit.
	ACCEPT BLOCK	An accept block icon is used to define a sequence of Ada code which appears as part of an accept statement, bounded with a "do" and an "end".
	EXCEPTION HANDLER	The exception handler icon represents the sequence of code within an exception handler, and is labeled with the exceptions being handled.
	LIBRARY PACKAGE	The library unit icon is used to represent a reference to a library package within the Ada Runtime System or user-defined library package.

The annotations to unit icons are as follows:

ANNOTATION	NAME	COMMENT
	CONCURRENCY INDICATOR, EXPLOSION INDICATOR	A concurrent unit (i.e., a unit with tasks) is denoted by a "C" placed in the icon. A compound unit (i.e., a "C" which has lower-level diagrams) is denoted by placing an "X" in the PK.
	PORT	Unit entry points are specified by placing a port on the specific unit icon. A port may represent a procedure entry point or an accept statement type entry. The name and details of the entry point appear in the standard unit's details as referenced by the number within the port. A circular port represents a subprogram entry, and a square port represents a task entry.
		
	SHADING	Shading is used to represent unit icons that have been introduced on a previous diagram of which the internals currently are being defined.

Connections between Unit icons are made by Associative icons. Associative icons may represent subprogram calls, entry calls, context clauses, execution sequence, etc. as follows:

ICON	NAME	COMMENT
	CALL	Invocations are represented by an arrow in the direction of the execution control flow.
	SIGNAL	A synchronization signal or interrupt from the Hardware Support System is to be represented by a dashed arrow as shown, for example, activating the main procedure, or terminating a task. Optionally, a name can be placed on the signal. A signal stub may be used to represent internal program queues.
	DATA BUS	The data bus is used to illustrate an interface with an external entity.
	LEXICALLY ENCLOSED	The lexically included inclusive icon represents the physical encapsulation of a task statement, exception handler, etc.
	CONTEXT CLAUSE	The context clause is used to represent the importing of Ada library units.

Annotations to Ada relative icons provide further details on the interaction between Unit icons, and help define execution threads within the system.

ANNOTATION	NAME	COMMENT
	SEQUENCE	Sequence order is designated by numbered arrows (followed in any Associative icon).
	PARAMETER	Parameter icons are to be used in conjunction with a call. The direction of the data corresponds to the direction of the parameter icon in relation to the call.
EXCEPTION	EXCEPTION	The exception icon is used to specify propagation of an exception in conjunction with a call and a raise statement.
	RENDEZVOUS STIPULATIONS	A lettered diamond is used to designate rendezvous details. This icon is placed at the tail of the call icon to indicate the entry call type and at the head of the call to indicate the details of the accept. Based on the particular letter, the diamond may represent a delay alternative, entry index, etc. If more than one rendezvous annotation applies, additional diamonds may be placed on the applicable side of the entry call as follows:
	CONDITIONAL	
	TIMED	
	UNCONDITIONAL	
	FAMILY MEMBER CALL/FAMILY ENTRY ACCEPT	
	GUARD	
	DELAY	
	TERMINATE	
	SELECT	
	LOOP	A loop construct is represented with a sequence of invocation arrows associated with loop-iterative icons to indicate loop and/or exit conditions. Invocation arrows should be numbered to indicate sequence.

ANNOTATION	NAME	COMMENT
	FOR LOOP	
	WHILE LOOP	
	LOOP UNTIL	
	LOOP EXIT	
	DELAY	The delay icon represents a delay statement, and is associated with the delay time.
	IF	The IF icon is detailed with its associated conditions.
	CASE	The CASE icon is detailed with the conditions on which the case is made.
	SELECT	The SELECT icon is used to associate alternatives within a select statement.
	EXCEPTION SELECT	The EXCEPTION SELECT is used to associate exception alternatives within an exception handler.

The following icons are classified as Miscellaneous icons as they do not translate directly into Ada program units or code.

ICON	NAME	COMMENT
	EXTERNAL ENTITY	The external entity icon represents any object outside the bounds of one system (software or hardware) within a system diagram.
	SYSTEM	The system icon is used to represent the entire system being modeled on the context diagram.
	OFF-PAGE CALL	The off page call is used to represent a call to a unit icon not represented on the given diagram. The icon is labeled with parent name and the call being made. Rendezvous stipulations must be included as required.
	OFF-PAGE INVOCATION	The off page invocation is used to represent invocation from a unit icon on an alternate diagram. The icon is labelled with the calling unit. In the case of a task type definition, the icon would be labelled with a general 'caller'.

The icon set described above will be used to generate four specific conceptual views of a system: environmental, communication, structural, and physical.

The environmental view consists of a Context Diagram which describes the boundary between the software system under development and its environment, including externally imported code.

Figure 1 is an example of an ADT Context Diagram.

The communication view is used to illustrate the communication and interaction between the major Ada units within the system being modeled. The Ada Communication

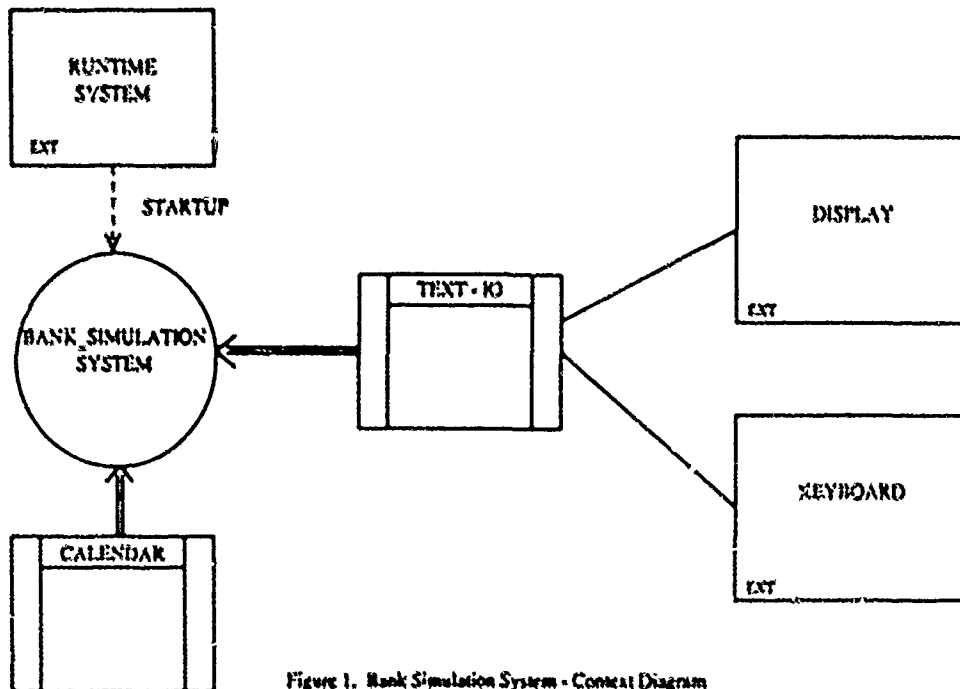


Figure 1. Bank Simulation System - Context Diagram

Diagram (ACD) will represent all program units which communicate through rendezvous, as well as the specific details of the rendezvous; calls to procedures within packages, including the specific details of the calls; and calls to major subprogram units, as well as the details of the calls.

The ACD, accompanied by the textual descriptions and Context Diagram, serves as an excellent tool for a top-level view of the system.

Figure 2 illustrates an example of an ACD.

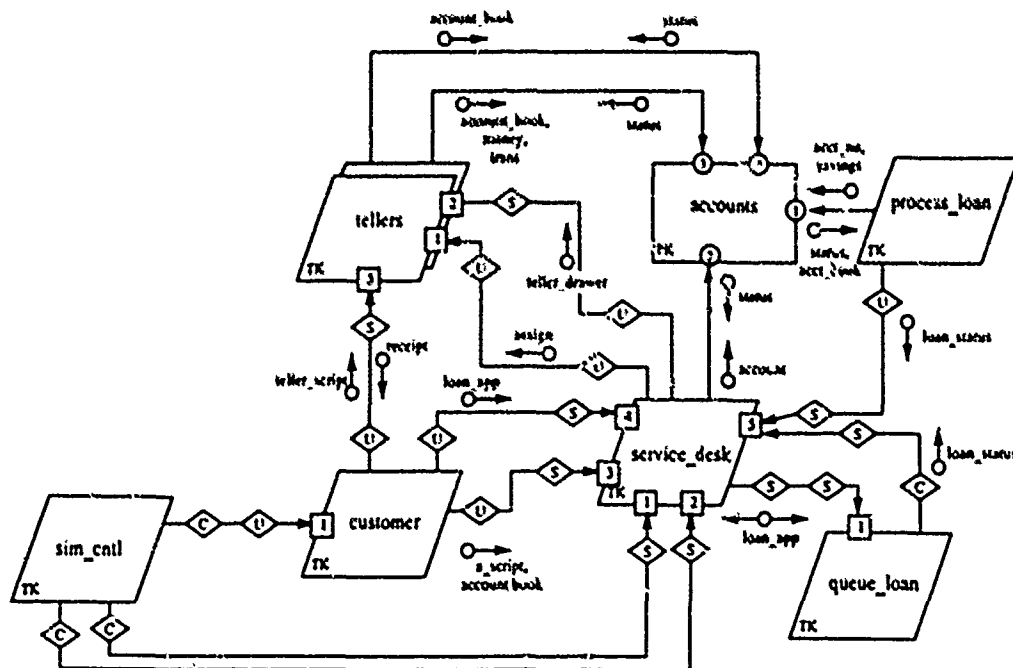


Figure 2. Bank Simulation System - Ada Communication Diagram

The internal logic, sequence, and structure within a program unit is represented by an Ada Structure Diagram (ASD). The ASD is similar to a Structure Chart, but has been extended to encompass Ada specific constructs and details. Accept statements, block structures, exception handlers, loops, cases, and if statements are all illustrated on an ASD. ASDs may be leveled to avoid over crowding. The users of the tool determine the level of detail shown on an ASD. The tool will generate source code for an Ada Unit based on level of detail entered.

Figure 3 is an example of an ASD.

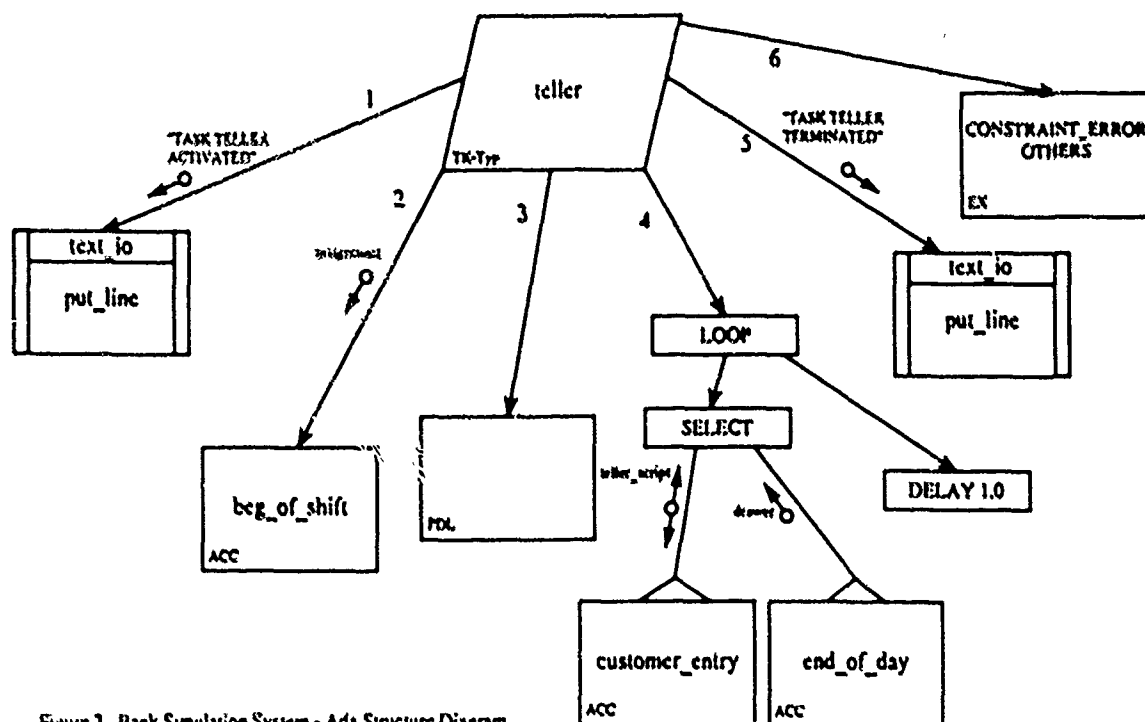


Figure 3. Bank Simulation System - Ada Structure Diagram

The physical view of the system is used to illustrate the compilation units and subunits and their file organization within the system through context clauses, sequence markers, and other annotations on a Physical Layout Diagram (PLD). PLDs enable the user to define and represent the file structure for storing the program elements within the application. This information is used by the source code generation function to determine the "withing" of the various program entities and by the Configuration Management Function for tracking the creation, modification, and access of compilation unit files. This diagram also aids the Software Engineer in partitioning the system to minimize recompilation.

Figure 4 illustrates an example of a PLD.

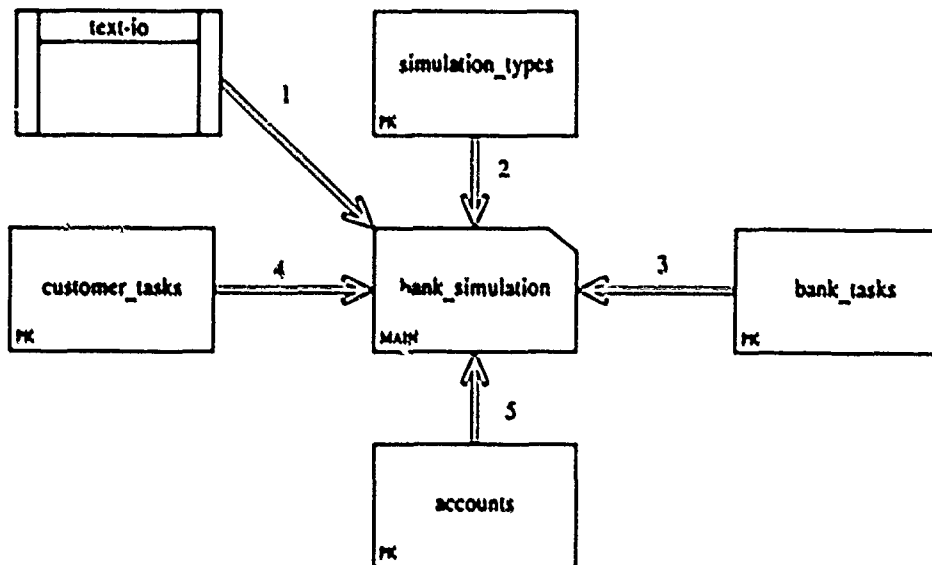


Figure 4. Bank Simulation System - Physical Layout Diagram

3.2 Textual Details of an Ada Design. A textual-details window is provided for each subprogram, package, task, and block unit icon within the system. Prompts are provided within the window to define information that is relevant to the corresponding Ada unit. Each unit's details window contains an explosion field to define the next lower level definition, a compilation unit field to specify the name of the file in which the unit is contained, a description field, a declarative section definition, and, if appropriate, a file for separately defined bodies. The user will be responsible for entering all information required to generate declarations and perform consistency checks between data definitions and usage. Information (such as identification and purpose, relationship to other units, and other definitive information) not contained in the ADT diagram that is used to satisfy the detailed design requirements should be specified in the description field. Each details window also includes fields dedicated to the unit or structure being defined (e.g., entry points, parameters, etc.). Examples of each program unit or structure textual details may be found in the right-hand column.

4.0 Validation

The ADT validation function consists of two main components: syntax checking and completeness-and-consistency analysis. The validation checks can be performed on the entire Ada design or any user-specified subset.

Syntax checks are performed on each type of diagram to ensure

SUBPROGRAM DETAILS

Name:	Name of subprogram
Subprogram Type:	Procedure, Function Main Procedure
Priority:	Applicable to main procedure only
Explosion Type:	ACD, ASD, PDL, etc.
Source Code File:	File in which Ada source is contained
Parent Unit:	Name of unit in which subprogram is contained (if applicable)
Parent Unit Type:	Type of parent unit (if applicable)
Formal Parameters:	
Declarative Section:	List of all objects and types defined within the declarative section of the subprogram body
Exceptions:	List of all exceptions handled and/or propagated within the subprogram
Description:	

that basic software engineering principles as well as Ada specific concepts are not violated.

The following are some of the major syntax checks:

- Every object is labeled and numbered correctly
- Free standing objects are not allowed
- There are no off-page Associative Icons
- Actual-parameters match formal parameters definitions
- Every External Entity is connected to the system directly, via a Context Clause, or indirectly, using a Library Package, via a Data Bus.

The completeness-and-consistency analysis ensures that all objects within the scope of this validation analysis are fully defined by their corresponding textual details and their usage is consistent with other objects in the design database.

5.0 ADT Source Code Generation

The ADT Source Code Generation facility will generate program templates in addition to detailed code. By analyzing ACDs and accompanying textual details, program templates (which include task specifications, package specifications, and rendezvous constructs) can be generated. ASDs and associated textual detail interpretation supply lower-level logic and sequence details to complete the Ada program. The use of the ADT Source Code Generator will eliminate syntax errors, and enforce the use of desired programming standards. Furthermore, changes in program design immediately can be reflected in the code.

6.0 Automatic Report and Document Generation

The ADT will provide a turn-key documentation facility from the design database. Analytical reports (such as Where Used and Data Dictionary reports) are generated to aid the software engineer in the creation of the Ada design. The hardcopy documents produced are formal Software Design Documents in compliance with DOD-STD-2167A. The document generator automatically provides section heading generation, page numbering, figure numbering, table of contents generation, security markings, figure cross-reference generation, and merging of text and figures on the same page. The final design documentation will be produced on a commercially available publishing system. The document generator turns out an intermediate design document containing publication commands for the publication system. A Customizer Kit will

TASK DETAILS	
Name:	Name of task:
Task Type:	Anonymous or Task Type
Number of occurrences:	
Explosion Type:	ACD, ASD, PDL, etc.
Priority:	
Representation Clause:	
Source Code File:	Name of file in which Ada source is contained
Master Program Unit:	Name of unit in which task is defined
Declarative Section:	List of all objects and types defined within the declarative section of the task body
Entry Points	1) Entry Name Parameters: 2) Entry Name Parameters: . . . n) Entry Name Parameters
Exceptions:	List of all exceptions processed and/or propagated within the task
Description:	

allow the user to tailor the format of the reports and documents to specific needs, making it flexible to other documentation standards.

SUMMARY

ADT is an Ada Design Tool which enables the Software Engineer to represent a top-level and detailed-level design in terms of the Ada language through the use of sophisticated graphical and textual editors. The tool can be used effectively to automate both the Object Oriented and Functional Decomposition Design approaches. In addition, ADT takes advantage of the richness of the Ada language, and supports the principles and goals of good software engineering practices.

The ADT is scheduled for Alpha test in April 1989 and Beta release in June 1989. Subsequent releases will contain the Ada Source Code Generator for the program structure as well as the detailed code.

REFERENCES

- [1] R.J. ...r, "System Design with Ada", Prentice Hall, 1984
- [2] Grady Booch, "Software Engineering with Ada", Second Edition, Benjamin/Cummings Publishing Company, Inc., 1987
- [3] Reference manual for the Ada Programming Language, ANSI/MIL-STD-1815a-1983
- [4] P. Ward and S. Mellor, "Structured Development for Real-Time Systems", Yourdon Press Computer Series, 1986
- [5] G. Cherry, "PAMELA Designers Handbook", Analytical Sciences Corporation, 1986
- [6] Department of Defense, "DoD-STD-2167A", 29 February 1988
- [7] McDonnell Douglas, "Ada Software Development Curriculum: Ada in Software Design", 1987
- [8] UNISYS User Manual, Version 3.2, Dept 2239, Unisys, September 22, 1988

BLOCK DETAILS

Name:	Name of Block
Block Type:	Standard, Accept, PDL, Exception
Explosion Type:	ACD, ASD, PDL, etc.
Parent Unit:	Name of unit in which block is contained
Parent Unit Type:	Type of unit in which parent is contained
Parent Unit Source File:	Name of file in which parent Ada source is contained
Declarative Section:	List of all data elements defined within the declarative section of the block
Exceptions:	List of all exceptions processed and/or propagated within the block
Description:	

PACKAGE DETAILS

NAME:	Name of Package
Explosion Type:	ACD, ASD, PDL, etc.
Source Code File:	Name of file in which Ada source is contained
Parent Unit:	Name of unit in which package is contained (if applicable)
Parent Unit Type:	Type of Parent Unit (if applicable)
Specification:	List of all objects and types defined in the specification of the package
Declarative Section:	List of all objects and types defined within the declarative section of the package body
Exceptions:	List of exception processed and/or propagated in the package body
Description:	

BIOGRAPHICAL SKETCHES

Mr. Tupper has worked in software tool development and real-time command-and-control systems for 15 years. As Engineering Section Head, he is responsible for development of a complete software engineering environment which is targeted to support project development from requirements specification through unit testing. Previously, Mr. Tupper worked on several real-time simulators and command/control systems. The most recent was the Spanish Navy program where he was the Senior Engineer responsible for design and development of Simulation Support Processor software for the land-based testing of FFG- and CV-class naval ships. Mr. Tupper received his bachelor's and master's degrees in Computer Science from Pratt Institute. He published several IR&D reports and papers on automated software development tools and environments, and operating systems development.



Ms. Levitz has 22 years of experience in the design, documentation, implementation, and test of software systems and software support. Her primary emphasis has been in development of embedded real-time operating systems for military applications. She was responsible for design and development of the Memory Processor Operating System (MPOS) on the Trident II navigational subsystem shipboard computers. Ms. Levitz is a leader of a group developing the Embedded Multiprocessing Ada Runtime Support System (E-MARS), and is the author of five operating system-related publications. She earned a bachelor's degree in Mechanical Engineering from the University of Hartford and a master's degree in mechanical engineering and applied mathematics from Rensselaer Polytechnic Institute. Ms. Levitz was a 1987 winner of the Unisys Excel Award for her work on the Trident II navigational subsystem operating system.



Ms. Hetzron has more than eight years of software engineering experience, having designed, implemented, and tested major program components for the SEALITE, HELF and TACDEW, Command Control and Communication applications. She also designed and implemented software functions for the Memory Processor Operating System (MPOS) used in the Trident II Navigational Subsystem. As a member of the Unisys Ada Core Group, she participated in development of the Embedded Multiprocessing Ada Runtime Support System and development of the Software Design Tool (SDT) and Ada Design Tool (ADT) within the Unisys Software Engineering Toolset (UNISET). She earned her BA in Computer Science from Queens College in 1980 and her MS in Computer Science from Polytechnic Institute of New York in 1983.



Ms. Barlev has six years of software engineering experience. She designed, implemented and tested extensions to the utility package (UPAK) for the Mk 92 Fire Control System. Ms. Barlev also designed and implemented a code auditor to detect specified program anomalies, and actively participated in the prototyping of Requirements Tools on the Lisp Machine. Currently, she is responsible for the graphical editors on the SUN workstation for the Requirements Specification Tool (RST) and the Ada Design Tool (ADT) within the Unisys Software Engineering Tool (UNISET). Ms. Barlev received her BA in Computer Science from Queens College in 1983, her MBA in Management Information Systems from Hofstra University in 1986, and currently is completing her Master's in Computer Science from Queens College.



Ms. Davanzo has five years of software engineering experience. She worked in a software development group for Trident I and Trident II Post-Mission Data Evaluation which included design, implementation, and testing software on various systems. These activities included conversion of Fortran programs into Ada on different computer systems. Ms. Davanzo currently is involved in the definition and development of the Ada Design Tool (ADT) which involves the Details Editor within the Unisys Software Engineering Toolset (UNISET). She earned her BS in Math with Computer Science from St. John's University in 1983 and her MA in Computer Science from Queens College in 1987.



OBJECTS WITH MULTIPLE REPRESENTATIONS IN ADA

K. M. George

Jag Sodhi

Oklahoma State University
Stillwater, Oklahoma

TELOS Federal Systems
Lawton, Oklahoma

ABSTRACT

Ada provides the facility for the separation of specification and implementation. However, there is a one-to-one correspondence between a specification and its implementation (if one exists). This paper presents an implementation scheme which provides multiple representations of an object. It is possible to choose an implementation from the available implementations. The user of the specification need not be concerned with the details of implementation.

1.0 INTRODUCTION

A programming paradigm such as object oriented programming which permits separation of specification and implementation provides several advantages. Specification provides an abstraction from computation [LIGU 86] which allows the use of complex objects without having to be concerned with their implementation. The user of a specification is spared from the implementation details. The burden of the appropriate choice of data structures is handled in the implementation and can be postponed as long as necessary. However, once the implementation chooses a data structure the user has no other choice. Modern programming languages incorporate this concept in their design and provide adequate facilities for separation of specification and implementation. In particular, Ada [VAXA] supports this separation of specification and implementation in generics, packages, tasks etc. The implementation bundles together data structures and the operations on the data structures. The specification provides the view of an object and a set of operations on it. Put another way, the specification provides the view of an abstract data type [CAWE 85]. There is a one-to-one correspondence between the specification and implementation. The user of the specification has no control over the choice of data structures and algorithms used in the implementation.

In some applications, efficiency might depend on the choice of data structures and algorithms. In such cases, the facility to select an implementation will be useful. As an example, consider a linear programming problem. The type of efficient representations are different for large sparse matrices and small matrices. The particular choice will depend on the application. So, ideally, one would like to have a linear programming package with the flexibility to choose efficient representation. The issue, then, is to provide specifications of abstract data types together with the capability to choose appropriate implementation. Such specification should not diminish the advantages of the separation of specification and implementation. This paper is concerned with an Ada solution to multiple representations of abstract data types and the means to choose an implementation. The user of the specification still need not be concerned with implementation details. The next section provides a definition of abstract data type, an example

of the problem and an Ada solution. The complete Ada program and output from a trial run are given in the appendix. Some side effects of the solution are discussed in section 3; and section 4 is the conclusion.

2.0 ABSTRACT DATA TYPE

An abstract data type (ADT) consists of a set of objects and a set of operations characterizing the behavior of the objects [LIGU 86]. The set of objects can be defined using relations of the operations. The specification of the ADT provides the name of the ADT and the names of the operations. The details of the operations are hidden in the implementation. An implementation of an ADT is called a realization of the ADT. An ADT can have more than one realization. An example is given in the next section.

2.1 EXAMPLE

Let us examine the ADT "stack" and two of its possible implementations. The ADT stack is defined by a set of functions and a set of defining relations. The implementations using an array and a linked list are examined.

abstype STACK is

```
functions :  
  PUSH : STACK x OBJECT --> STACK  
  POP : STACK --> STACK  
  TOP : STACK --> OBJECT
```

```
relations:  
  POP ( PUSH ) = id  
  PUSH ( POP, TOP ) = id if STACK is not  
    EMPTY
```

end STACK

The implementations must obey the defining relationships of the ADT. Two implementation schemes are shown below:

```
1) array-implementation is  
  /* using scalar I and array A */  
  I <- 0;  
  PUSH:  
    I <- I+1; A[I] <- X; /* X is an object */  
  POP:  
    I <- I-1;  
  TOP:  
    value of A[I];  
end array-implementation;
```

```

2) linked-list-implementation is
/* using HEAD as pointer to the top of the list, and
   a structure LIST with two components INFO and
   LINK */
HEAD <-- NULL;
PUSH:
    TEMP <-- new LIST /* TEMP is a temporary
                       pointer to a cell */
    TEMP.INFO <-- X; TEMP.LINK <-- HEAD;
    HEAD <-- TEMP;
POP:
    HEAD <-- HEAD.LINK;
TOP:
    value of HEAD.INFO;
end linked-list-implementation;

```

In the above implementation schemes, the data structures array, the linked list, and both persistent scalar variables I and HEAD are internal to the implementation. The operations PUSH, POP and TOP are the only names visible to outside users. It is easy to verify that both implementation schemes satisfy the defining equations. It is also obvious that the implementations of functions are directly related to the data structure chosen for the implementation. When an implementation is abstracted information is lost. In order to be able to choose a specific implementation, relevant information must be preserved by the abstraction. So, in this paper we examine an abstraction process which preserves the relevant information.

2.2 REALIZATION AS A TYPE

An ADT can be implemented using several data structures. We call an implementation of an ADT "t" a realization of t. A realization essentially is a data structure. The definition of a data structure is given by Reingold and Hansen [REHA 83] is shown below.

A data structure consists of three components:

- 1) a set of function definitions,
- 2) a storage structure, and
- 3) a set of algorithms, one for each function.

For our discussion, an algorithm is identified with its function name. Therefore, we can view a data structure as a pair (S,F) where S represents a storage structure and F represents a set of functions. An equivalence class structure can be imposed on a set of data structures. Let D be a set of data structures. Two data structures $d_1 = (S_1, F_1)$ and $d_2 = (S_2, F_2)$ in D are equivalent if d_1 and d_2 are realizations for the same set of ADTs. If D consists of exactly one equivalence class and if the equivalence is imposed by an ADT t, then D is called a realization class for t. In other words, every member of D will be an implementation of the ADT t. The set D contains more information (eg. storage structure) than the ADT t itself. If the information hidden in D is available to the user of t, then that user will be able to choose the appropriate realization based on the application from D. This can be achieved by viewing D as a type. We call this type realization. Using D and t, we can define a new type which allows one to select a realization for an ADT. This new type is called rabstype where, as an ADT, it is called abstype. The formal definitions are given below:

- 1) TYPE is realization
VALUES are in D
OPERATION is selection: $D \rightarrow d$, where d is a member of D.

- 2) TYPE is rabstype
VALUES are in t x D
OPERATION is binding: $(t,d) \rightarrow t.d$
where t.d refers to an instance of t with realization d.

An Ada package specification for the ADT stack is given in the next section. The operations, selection and binding, defined above will correspond to an instantiation of a generic package. The type rabstype is implemented as a generic package specification.

2.3 Ada IMPLEMENTATION

In order to implement the above concepts in Ada, the realization type should be visible through the specification. The following package specification provides a specification:

```

package STACK_IMPLEMENTATION_LIST is
type LIST_OF_STACK_IMP_TYPE is
    (ARRAY_IMP, LINKED_LIST_IMP);
generic
    STACK_IMP : LIST_OF_STACK_IMP_TYPE;
package STACK is
    procedure PUSH (OBJECT : INTEGER);
    procedure POP;
    function TOP return INTEGER;
end STACK;
end STACK_IMPLEMENTATION_LIST;

```

The list of available implementations are made visible as the values associated to the enumeration type LIST_OF_STACK_IMP_TYPE. The choice of an implementation is the same as instantiation of the generic package STACK using the appropriate value. For the complete implementation and a test run, the reader is referred to the appendix. Some of the advantages and disadvantages related to this approach are discussed in the next section.

3.0 SIDE EFFECTS OF THE APPROACH

The method illustrated by the specification in the previous section can be used to construct interfaces to reusable software. The specification can be viewed as a window as shown in figure 1.

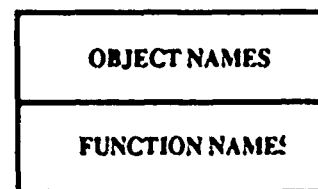


Figure 1

Object names refer to implementation type and the function names refer to a function or a procedure. For example, assume that there are several procedures which implement sorting algorithms such as bubble sort, quick sort etc. Then the function name can be sort and object names can be bubble, quick etc. The advantage is that the user of the specification needs to be concerned only with the algorithm type and the name of the function. The user need not know which

packages to use or what specific name to use and so on. The disadvantage, however, is the inefficiency associated to generic instantiation. The runtime inefficiency probably can be removed by appropriate substitutions at compile time if such capability exists.

4.0 CONCLUSION

In this paper we have addressed the issue of multiple implementations and dynamic implementation choice of abstract data types in Ada. The method used is illustrated using an example, a complete Ada program. The separation of specification and implementation is not compromised. In order to provide a choice of implementation, a level of abstraction is chosen such that the names of implementation are visible.

REFERENCES

[CAWE 85] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", Computing Surveys, Vol. 17, No. 4, December 1985.

[LIGU 86] B. Liskov and J. Guttag, Abstraction and Specification in Program Development, The MIT Press, Cambridge, Massachusetts, 1986.

[MALE 86] M. Marcotty and H. F. Lodgard, Programming Language Landscape: Syntax/Semantics/Implementation, Second edition, SRA, Chicago, 1986.

[REHA 83] E. M. Reingold and W. J. Hansen, Data Structures, Little, Brown and Company, Boston, 1983.

[VAXA] Vax Ada Language Reference Manual.

APPENDIX

```
-- This package provides the list of available implementations of the abstract
-- data type STACK.
```

```
package STACK_IMPLEMENTATION_LIST is
```

```
type LIST_OF_STACK_IMP_TYPE is (ARRAY_IMP, LINKED_LIST_IMP);
```

```
    -- List of implementations.
```

```
generic
```

```
    STACK_IMP : LIST_OF_STACK_IMP_TYPE; -- Implementation type to be chosen.
```

```
-- Abstract data type follows:
```

```
package STACK is
```

```
    procedure PUSH (OBJECT:INTEGER);
```

```
    procedure POP;
```

```
    function TOP return INTEGER;
```

```
end STACK;
```

```
end STACK_IMPLEMENTATION_LIST;
```

```

-- This package body contains all implementations including the subprograms
-- associated to abstract data type STACK.

with TEXT_IO; use TEXT_IO;
package body STACK_IMPLEMENTATION_LIST is

    package INT_IO is new INTEGER_IO(INTEGER);
    use INT_IO;
    type ARRAY_TYPE is array (1..100) of INTEGER; -- storage for array
                                                    -- implementation.

    type NODE;
    type LINK is access NODE;
    type NODE is
        record
            VALUE : INTEGER;
            NEXT : LINK;
        end record;
    STACK_ARRAY : ARRAY_TYPE;
    STACK_TOP   : INTEGER := 0; -- stack top for array
    LINK_TOP    : LINK := null; -- stack top for linked list

-- Array implementation of PUSH.
    procedure ARRAY_PUSH (OBJECT:INTEGER) is
    begin
        STACK_TOP := STACK_TOP + 1;
        STACK_ARRAY (STACK_TOP) := OBJECT;
-- Messages to verify the actions.
        PUT("ARRAY IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS");
        PUT(OBJECT);
        NEW LINE;
    end ARRAY_PUSH;

-- Linked list implementation of PUSH.
    procedure LINK_PUSH (OBJECT : INTEGER) is
        TEMP : LINK;
    begin
        TEMP := new NODE;
        TEMP.VALUE := OBJECT;
        TEMP.NEXT := LINK_TOP;
        LINK_TOP := TEMP;
-- Messages to verify the actions.
        PUT("LINK IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS");
        PUT(OBJECT);
        NEW LINE;
    end LINK_PUSH;

-- Array implementation of POP.
    procedure ARRAY_POP is
    begin
        STACK_TOP := STACK_TOP - 1;
-- Messages to verify the actions.
        PUT("ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE POPED IS");
        PUT(STACK_ARRAY(STACK_TOP+1));
        NEW LINE;
    end ARRAY_POP;

```

```

-- Linked list implementation of POP.
  procedure LINK_POP is
  begin
  -- Messages to verify the actions.
    PUT("LINK IMPLEMENTATION OF STACK IS USED AND VALUE POPPED IS");
    PUT(LINK_TOP.VALUE);
    NEW_LINE;
    LINK_TOP := LINK_TOP.NEXT;
  end LINK_POP;

-- Array implementation of the function TOP.
  function ARRAY_TOP return INTEGER is
  begin
  -- Messages to verify the actions.
    PUT("ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS");
    PUT(STACK_ARRAY(STACK_TOP));
    NEW_LINE;
    return STACK_ARRAY(STACK_TOP);
  end ARRAY_TOP;

-- Linked list implementation of the function TOP.
  function LINK_LIST_TOP return INTEGER is
  begin
  -- Messages to verify the actions.
    PUT("LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS");
    PUT(LINK_TOP.VALUE);
    NEW_LINE;
    return LINK_TOP.VALUE;
  end LINK_LIST_TOP;

-- The implementation of the abstract data type STACK. It uses one of the above
-- Defined implementations.
  package body STACK is
    procedure PUSH (OBJECT : INTEGER) is
    begin
      if STACK_IMP = ARRAY_IMP then
        ARRAY_PUSH(OBJECT);
      elsif STACK_IMP = LINKED_LIST_IMP then
        LINK_PUSH(OBJECT);
      else
        null;
      end if;
    end PUSH;

    procedure POP is
    begin
      if STACK_IMP = ARRAY_IMP then
        ARRAY_POP;
      elsif STACK_IMP = LINKED_LIST_IMP then
        LINK_POP;
      else
        null;
      end if;
    end POP;
  end package body STACK;

```



```

function TOP return INTEGER is
begin
  if STACK_IMP = ARRAY_IMP then
    return ARRAY_TOP;
  elsif STACK_IMP = LINKED_LIST_IMP then
    return LINK_LIST_TOP;
  else
    return -1000;
  end if;
end TOP;
end STACK;

end STACK_IMPLEMENTATION_LIST;

```

```
-- This a main program to test the package STACK which is a generic package
-- which implements the abstract data type STACK. It is possible to select the
-- implementation one likes from among the available implementations. This
-- programs tests the generic package STACK by instantiating with the two
-- possible values.
-- Since its purpose is to verify the relevant packages, there are no means
-- of handling exceptions.
```

```
with TEXT_IO; use TEXT_IO;
with STACK_IMPLEMENTATION_LIST;
use STACK_IMPLEMENTATION_LIST;
procedure STACK_CHECK is
  package INT_IO is new INTEGER_IO(INTEGER);
  use INT_IO;
begin
```

```
-- Test array implementation of STACK.
  declare
    package STACK_PACKAGE is new STACK(ARRAY_IMP);
    use STACK_PACKAGE;
    X,Y:INTEGER;
  begin
    for I in 1..5 loop
      PUT("INTEGER TO BE PUSHED :");
      GET(X);
      NEW_LINE;
      PUSH(X);
      Y := TOP;
    end loop;
    for I in 1..4 loop
      POP;
      Y := TOP;
      PUT("NEW TOP IS :");
      PUT(Y);
      NEW_LINE;
    end loop;
  end;
```

```
-- Test linked list implementation of STACK.
  declare
    package STACK_PACKAGE is new STACK(LINKED_LIST_IMP);
    use STACK_PACKAGE;
    X,Y:INTEGER;
  begin
    for I in 1..5 loop
      PUT("INTEGER TO BE PUSHED :");
      GET(X);
      NEW_LINE;
      PUSH(X);
      Y := TOP;
    end loop;
    for I in 1..4 loop
      POP;
      Y := TOP;
      PUT("NEW TOP IS :");
      PUT(Y);
      NEW_LINE;
    end loop;
  end;
```

SAMPLE OUTPUT

INTEGER TO BE PUSHED :	
ARRAY IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	100
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	100
INTEGER TO BE PUSHED :	
ARRAY IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	200
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	200
INTEGER TO BE PUSHED :	
ARRAY IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	300
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	300
INTEGER TO BE PUSHED :	
ARRAY IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	400
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	400
INTEGER TO BE PUSHED :	
ARRAY IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	500
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	500
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE POPED IS	500
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	400
NEW TOP IS :	400
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE POPED IS	400
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	300
NEW TOP IS :	300
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE POPED IS	300
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	200
NEW TOP IS :	200
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE POPED IS	200
ARRAY IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	100
NEW TOP IS :	100
INTEGER TO BE PUSHED :	
LINK IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	600
LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	600
INTEGER TO BE PUSHED :	
LINK IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	700
LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	700
INTEGER TO BE PUSHED :	
LINK IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	800
LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	800
INTEGER TO BE PUSHED :	
LINK IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	900
LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	900
INTEGER TO BE PUSHED :	
LINK IMPLEMENTATION OF PUSH IS USED AND VALUE PUSHED IS	1000
LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	1000
LINK IMPLEMENTATION OF STACK IS USED AND VALUE POPED IS	1000
LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	900
NEW TOP IS :	900
LINK IMPLEMENTATION OF STACK IS USED AND VALUE POPED IS	900
LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	800
NEW TOP IS :	800
LINK IMPLEMENTATION OF STACK IS USED AND VALUE POPED IS	800
LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	700
NEW TOP IS :	700
LINK IMPLEMENTATION OF STACK IS USED AND VALUE POPED IS	700
LINK IMPLEMENTATION OF STACK IS USED AND VALUE OF TOP IS	600
NEW TOP IS :	600

K. M. GEORGE has a Ph.D. in Mathematics from the State University of New York, Stonybrook. He is an Associate Professor, Department of Computing and Information Sciences, Oklahoma State University, Oklahoma. He has published several papers in functional and object oriented programming.



JAG SODHI has a Master Degree in Mathematics, a Degree in Telecommunication Engineering, and is a Graduate of IBM in Data Processing. He has many years of Data Processing experience in business, financial and scientific applications in various EDP machines and languages. He has conducted numerous professional classes and seminars on these subjects. His publishing credits include numerous training courses on Ada and software engineering. Jag is a senior system engineer and in charge of education and training at TELOS Federal Systems, Region I.

A Software Development Tool Using Ada -- Pseudo Code Management System

By Dar-Biau Liu

Department Of Computer Science And Engineering
California State University, Long Beach
Long Beach, CA. 90840

ABSTRACT

This software development tool will allow for completion of one of the detailed design specifications, contributing to the architectural design and the detailed design modularization of the software product. The interface specifications for the various modules are incorporated within a "Module Template". This Module Template and the pseudo code body can later be expanded into source code. The PCMS will allow system designers to document and track software modules more effectively as they are being created. It also has the capability to reconstruct a structured chart graphically for a given subsystem.

I. INTRODUCTION

During the development of a software product, it is helpful to provide various tools and mechanisms whereby the programmers working on the product can document the development cycles and identify the various inter-connecting paths between modules. This facilitates the easy generation and maintenance of robust, efficient codes.

Our goal is to design such a software development tool to aid in the design of a software product. This tool will allow for completion of one of the detailed design specifications, contributing to the architectural design and the detailed design modularization of the software product.

One of the area that such a tool is advantageous is during the initial design of the interface specifications for the various modules written for the subsystem of the product. These specifications are often incorporated

within a "Module Template". An example of such a template is shown in section II. This template ultimately leads to the creation of the pseudo-code body.

This module template and the pseudo-code body can later be expanded into source code. Modules interface specifications such as the Module Template are effective notations for architectural design when they are used in combining with structured charts and data flow diagrams. They also provide a natural transition from architectural design to detail design, and from detail design to implementation.

The development of any software product is an expansive and time-consuming process. The need for accurate, well-documented, maintainable source code is evidenced by the growing amount of emphasis placed on fault-tolerant, large scale, efficient system packages. A tool which provides the ability to fully document and track the development process of the various modules within a product is one of the several important aids which is needed to cope with the increasingly difficult environment within which software packages are currently being developed.

The Pseudo Code Management System (PCMS) will allow system designers (or Software Engineers) to more effectively document and track software modules as they are being created. The system fits well within the philosophy of "Top-Down" development methodology by providing a valuable tool to document, verify completeness and maintain an archivable tracking method for the product under review.

The life-cycle model was used to manage the development of PCMS. The development, operating, and maintenance of PCMS will be done on VAX 8530 using the VMS operating system. DEC Ada Compiler is chosen to develop the system.

The basic features of PCMS are as follows:- It has

- On-line capabilities to create, delete, and modify on each entry in each module template.
- Capability to print an entire module template for a given module.
- Capability to print only the most recently changed entries.
- Capability to reconstruct a structured chart graphically for a given subsystem (from the informations provided by the module templates in the subsystem).

II. MODULE TEMPLATE STRUCTURE

A) For each module, there corresponds a module template which includes the following entries:-

MODULENAME : (module name or number)
 PARTOF : (subsystem name or number)
 CALLED BY : (module name or number)*
 PURPOSES : (textual description)
 DESIGNER/DATE : (designer and date)
 PARAMETERLIST : (names, modes, attributes)*
 INPUTASSERTION : (preconditions)
 OUTPUTASSERTION : (postconditions)
 CALLINGFOR : (module name or number)*
 GLOBALS : (name, mode, shared with)*
 SIDE-EFFECTS : (textual description)
 LOCALDATA : (name, mode, attributes)*
 EXCEPTIONS : (conditions, responses)*
 TIMINGCONSTRAINTS : (description)
 OTHERLIMITATIONS : (description)
 MODULEBODY : (pseudo-code)
 MODIFIEDBY : (whom, when, what, why)*

* denotes zero or more occurrences of the entities enclosed in parentheses.

B) Module Template entry definitions:

1. MODULENAME:
 Attributes : limited to 10 characters or numbers
 Definition : The module name identifies the module template. All references to this module in other modules should use this name to identify this module template.
2. PARTOF:
 Attributes : limited to 10 characters or numbers
 Definition : This entry identifies which subsystem this module is part of.

3. CALLED BY:
 Attributes : limited to 10 characters or numbers
 Definition : This entry identifies all the modules which call this module.

4. PURPOSES:
 Attributes : textual description
 Definition : A short abstract describing the functional purpose of the module.

5. DESIGNER/DATE:
 Attributes : The designer name is limited to 20 characters and date is of the form MM/DD/YY.

Definition : This entry denotes the initial author and date of the template.

6. PARAMETERLIST:
 Attributes : names - limited to 10 characters
 modes - limited to IN, OUT, or INOUT
 attributes limited to 15 characters
 Definition : This entry identifies all parameters.

7. INPUTASSERTION:
 Attributes : textual description
 Definition : This entry details the conditions that should exist prior to calling this module.

8. OUTPUTASSERTION:
 Attributes : textual description
 Definition : This entry details the conditions that exist upon leaving this module.

9. CALLINGFOR:
 Attributes : limited to 10 characters or numbers
 Definition : This entry identifies all the modules which are called by this module.

10. GLOBALS:
 Attributes : names - limited to 10 characters
 attributes - limited to 20 characters
 Definition : This entry identifies all global variables used by this module.

11. SIDE-EFFECTS:
 Attributes : textual description
 Definition : This entry details the possible side-effects of executing the module.

12. LOCALDATA:

Attributes : names - limited to 10 characters
attributes - limited to 20 characters

Definition : This entry identifies all local variables used by this module.

13. EXCEPTIONS:

Attributes : both conditions and responses are textual

Definition : This entry identifies conditions that would cause an exception or error to arise and the assigned response for that exception.

14. TIMINGCONSTRAINTS:

Attributes : textual description

Definition : This entry identifies the time it should take for processing.

15. MODULEBODY:

Attributes : textual description

Definition : This entry lists the actual processing of the module in a stepwise manner using pseudo code

16. OTHERLIMITATIONS:

Attributes : textual description

Definition : This entry lists the functional limitations of the module which are not already specified.

17. MODIFIEDBY:

Attributes : whom - limited to 20 characters

when - MM/DD/YY

what - limited to 30 characters

why - limited to 20 characters

Definition : This entry is the history of the module.

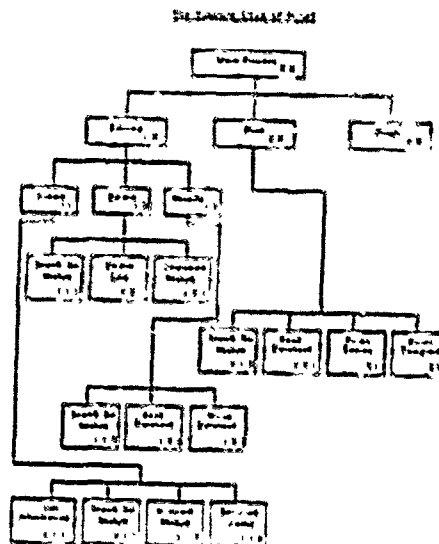


Figure 1

With Text_10;
With Direct_10;

Package PCMS is

```
subtype NAME_TYPE is string(1..80);
subtype DESC_TYPE is string(1..20);
type DATE_TYPE is new string(1..80);
type PARM_NAME_TYPE is
    new string(1..80);
...
NUMBER_CALLED : constant := 20;
NUMBER_PARM : constant integer := 20;
NUMBER_GLOBALS : constant integer := 20;
NUMBER_LOCALS : constant integer := 20;
NUMBER_EXCEPTIONS : constant integer := 20;
NUMBER_MODIFIED : constant integer := 20;
MAX_MODULES : constant integer := 20;
HASH_TABLE_SIZE : constant integer := 20;
NUMBER_SUBMODULES : constant integer := 20;
```

...

```
type MODULE_TEMPLATE is record
    NAME : NAME_TYPE;
    INDEX : INTEGER;
end record;
```

```
type MODULE_ARRAY is array(count)
of MODULE_TEMPLATE;
```

```
type MODULE_DESC_RECORD is record
    NAMES : NAME_TYPE;
    MODES : NAME_TYPE;
    ATTRIBUTES : NAME_TYPE;
    PURPOSES : DESC_TYPE;
end record;
```

III. SYSTEM ARCHITECTURAL DESIGN:-

HIPO Diagrams are used to represent the system architecture. The overview of the system is as shown in structured chart Figure 1.

Each Module Template is considered as a record in a direct access file. In order to facility the access of the direct access file, a look-up hash table is also established with the module name and the index.

```

type LOCAL_LIST_RECORD is record
  NAMES : NAME_TYPE;
  ATTRIBUTES : NAME_TYPE;
  PURPOSES : DESC_TYPE;
end record;

type GLOBAL_LIST_RECORD is record
  NAMES : NAME_TYPE;
  MODES : NAME_TYPE;
  PURPOSES : DESC_TYPE;
  SHARED_WITH : NAME_TYPE;
end record;

type EXCEPTION_RECORD is record
  CONDITIONS : DESC_TYPE;
  RESPONSES : DESC_TYPE;
end record;

type MODIFIED_BY_RECORD is record
  WHOM : NAME_TYPE;
  DATE : NAME_TYPE;
  WHAT : DESC_TYPE;
  WHY : DESC_TYPE;
end record;

type EXCEPTION_MATRIX is array(1..
  NUMBER_EXCEPTIONS) of EXCEPTION_
  RECORD;

type LOCALS_MATRIX is array(1..
  NUMBER_LOCALS) of LOCAL_LIST_
  RECORD;

type GLOBALS_MATRIX is array(1..
  NUMBER_GLOBALS) of GLOBAL_LIST_
  RECORD;

type PARM_MATRIX is array(1..
  NUMBER_PARM) of MODULE_DESC_
  RECORD;

type MODIFIED_BY_MATRIX is array(1..
  NUMBER_MODIFIED) of MODIFIED_BY_
  RECORD;

type CALLED_BY_MATRIX is array(1..
  NUMBER_SUBMODULES) of MODULE_
  TEMPLATE;

type CALL_FOR_MATRIX is array(1..
  NUMBER_SUBMODULES) of MODULE_
  TEMPLATE;

type UNIT_RECORD is record
  MODULE_NAME : NAME_TYPE;
  SUBSYSTEM_NAME : NAME_TYPE;
  CALLED_BY : CALLED_BY_MATRIX;
  PURPOSES : DESC_TYPE;
  DESIGN DATE : NAME_TYPE;
  PARM_LIST : PARM_MATRIX;
  LENGTH_PARM_LIST : integer;
  INPUT ASSERTION : DESC_TYPE;
  OUTPUT ASSERTION : DESC_TYPE;
  CALL FOR : CALL_FOR_MATRIX;
  GLOBALS : GLOBALS_MATRIX;
  LENGTH_GLOBALS_LIST : integer;
  SIDE_EFFECTS : DESC_TYPE;
  LOCAL_DATA : LOCALS_MATRIX;
  LENGTH_LOCALS_LIST : integer;
  EXCEPTIONS : EXCEPTION_MATRIX;

```

```

  TIMING_CONSTRAINTS : DESC_TYPE;
  LIMITATIONS : DESC_TYPE;
  PSEUDO_CODE : DESC_TYPE;
  MODIFIED BY : MODIFIED_BY_MATRIX;
  LENGTH_MOD_MATRIX : integer;
end record;

```

```

Data_Base_Rec : UNIT_RECORD;

type Record_Index is range 1.. MAX_MODULE;

type Unit_Rec_Vect is array(Record_Index)
  of UNIT_RECORD;

Data_Base_Vect : Unit_Rec_Vect;

package Database_IO is New Direct_IO(
  UNIT_RECORD);

Use Database_IO;

Data_Base_File : file_type;
f : Database_IO.file_type;

package INT_IO is New Integer_IO(integer);

Use INT_IO;

type LOOKUP_TABLE is array(1.. MAX_
  MODULES) of MODULE_TEMPLATE;

HASH_TABLE : LOOKUP_TABLE;

.....

Procedure Display_Menu
  (Menu_Number : IN NAME_TYPE);

Procedure Display_Modules
  (Root_Module_Name : IN NAME_TYPE);

Procedure Print_Modules;

Procedure Initialize_Data_Base;

Procedure Create_Module
  (Module_Name : IN NAME_TYPE);

Procedure Delete_Module
  (Module_Name : IN NAME_TYPE);

Procedure Modify_Module
  (Module_Name : IN NAME_TYPE);

Procedure Search_For_Module
  (Module_Name : IN NAME_TYPE;
  Hash_Index_Number : OUT integer;
  DataBase_Index : OUT integer);

Procedure Find_Sub_Modules
  (Module_Name : IN NAME_TYPE;
  Call_By_Vect : OUT MODULE_TEMPLATE;
  CallFor_Vect : OUT MODULE_TEMPLATE);

```



```

Procedure Allocate_Module
  (Module_Name: IN NAME_TYPE;
   Index_Number: OUT Integer);

```

```

Procedure Deallocate_Module
  (Index_Number: IN Integer);

```

```

Procedure Read_Data_Base
  (Location: IN Integer);

```

```

Procedure Write_Data_Base
  (Location: IN Integer);

```

```

....

```

```

end PCMS;

```

The detail design of the major functions for the PCMS will be described in pseudo-codes in the next section.

IV. DETAIL DESIGNS

1.1 Editing subsystem:

Procedure Create_Module

```

begin
  Search_For_Module
  if module does not exist then
    begin
      Get_Information
      Allocate_Space_For_Module
      Set_Link_Called
      Write_Record to database file
    end
  else
    print error message
  end if
end;

```

Procedure Modify_Module

```

begin
  Search_For_Module
  if module does not exist then
    print error message
  else
    begin
      read module into buffer record
      display menu
      read the chosen field from the
        buffer record
      display the field on crt
      enable editing
      save the modifications
      replace the modified field to the
        database file
    end
  end if
end

```

Procedure Delete_Module

```

begin
  Search_For_Module
  if module does not exist then
    print error message
  else
    begin
      read module into a buffer record
      from database file
      assign CALLED_BY field to a
        temporary location
      assign CALL_FOR field to a
        temporary location
      DEALLOCATE_MODULE from the hash table
      DEALLOCATE_CALL_BY_REFERENCE
      DEALLOCATE_CALL_FOR_REFERENCE
    end
  end if
end

```

Procedure Search_For_Module

```

begin
  Initialize variable RESULT to zero
  search the hash table until a module
    with given name found
  if found then assign the position
    where it was found to RESULT
    and then exit
end

```

Procedure Allocate_Space_For_Module

```

begin
  Initialize RESULT to zero
  if TABLE_SIZE greater than MAX_MODULE
    allowed then raise exception
  else assign the FIRST_AVAILABLE_POS
    to RESULT
  end if
  put the index in the corresponding
    hash table and the MODULE_NAME in
    the name field of the hash table
  search the hash table until the next
    first available position found
  assign that value to
    FIRST_AVAILABLE_POS
end

```

Procedure Set_Link_Called

```

begin
  assign CALLED_BY, CALL_FOR and the
    length of each field to a temporary
    location

  for all the element in the CALLED_BY
    field loop

    get the record corresponding to the
      element from database file
    search through its CALL_FOR field
      until an empty position found
    assign the file index of current
      module template to that index
      position
  end loop
end

```

```

    write the record back to the file
end loop

for all the element in the CALL_FOR
  field loop

  get the record corresponding to
  the element from database file
  search through its CALLED_BY field
  until an empty position found
  assign the file index of the current
  module template to that position
  assign the module name to the name
  field of that position
  write the record back to the file

end loop
end

```

Procedure Get_Information

```

begin
  get information of each field of the
  module template from the screen
  interactively
end

```

2.0 Printing subsystem

Procedure Print_A_Module

```

begin
  print each field of the module template
  one by one
end

```

Procedure Print_Modified_Entries

```

begin
  print all the fields that have been
  changed by checking each field flag
end

```

3.0 Display subsystem

Procedure Draw_Box

```

begin
  calculate the row and column of
  the module box
  draw the module box
  if the module calls itself then
    draw the recursive module lines
end

```

Procedure Draw_Connection_Lines

```

begin
  calculate row of the connection line
  if more than one callees then
    calculate the columns of the first
    and last callees of the module
  end
end

```

```

  draw the horizontal line connecting
  the first callee and last callee
  draw vertical lines for each callees
else
  draw a vertical line connecting the
  module and callee
end if
end

```

Procedure Draw_Module_Name_List

```

begin
  for all the modules loop
    draw the module number
    draw the module name
  end loop
end

```

Procedure Draw_A_Tree

```

begin
  for the number of queue items loop
    Draw_Box
    if there are callees then
      Draw_Connection_Lines
    end if
    keep record of the last row number
  end loop
end

```

Procedure View_and_Move

```

begin
  initialize first row and first column
  loop
    Display_On_Screen
    display move choice and get choice
    if choice is to exit then exit loop
  else
    change first row and column accord-
    ing to the choice
  end loop
end

```

Procedure Find_Recursive_Module

```

begin
  for all modules loop
    find MODULE_INDEX
    for all callees of the module loop
      if the module calls itself then
        mark the module recursive
      end loop
    if the module is recursive then
      remove the recursively called
      callee from the callees table
    end loop
  end
end

```

Procedure Up_Tracing

```

begin
  initialize the up tracing queue index
  to the rear of the queue
  while the up tracing queue index is

```

```

not the top of the tree loop
Initialize the left tracing queue
index to the left pointer of the
up tracing queue index
while module pointed by the left
tracing queue index exists loop
    add the left most callee's
    relative column position of the
    module pointed by the rear queue
    index to the relative position
    of the module pointed by the
    left tracing queue index
    move the left tracing queue
    index to the left
end loop
Initialize the right tracing queue
index to the right pointer of the
up tracing queue index
while module pointed by the right
tracing queue index exists loop
    add the right most callee's
    relative column position of the
    module pointed by the rear queue
    index to the relative position of
    the module pointed by the left
    tracing queue index
    move the left tracing queue index
    to the left
end loop
move the up tracing queue index to
the caller
end loop
end

```

```

Procedure Cal_Absolute_Module_Position
begin
    for all the queue items from rear
    to front loop
        if the module has caller then
            add its caller's absolute column
            position which has been calculated
            before to this module's relative
            column position to make the column
            position absolute
        else no calculation,
            the relative column position is
            also its absolute column position
        end if
    end loop
end

```

```

Procedure Build_Up_Queue
begin
    initialize queue front and rear to 1
    the layer position of the first queue
    item is 1
    assign the module index and module

```

```

number of the root module to the
first queue item
while the rear of the queue is not
equal to the front of the queue
loop
    for all the callees of the module
    pointed by the rear of the queue
    loop
        move the front of the queue by 1
        assign the module index, module
        number and recursive information
        of the callee to the front
        queue item
        calculate the layer and column
        position of the front queue
        item
        calculate the up, left, right
        pointer of the front queue item
        the down pointer of the rear
        queue item points to its
        first callee
    end loop
    Up_Tracing to make room for the
    newly added callees
    move the queue rear by one
end loop
reset the rear of the queue to 1
Cal_Absolute_Module_Position
end

```

```

Procedure Display_Module_List_And_
Get_Root_Module_Index
begin
    calculate rows and the number of
    modules on the last row
    display the module's module number
    and module name
    prompt the user to get the root
    module number
    calculate the root module's module_
    index
end

```

```

Procedure Read_Data_Base_File
begin
    assign the local hash table with the
    values of the global hash table
    prompt the user to enter the file name
    get the file name
    read the number of the modules
    for the number of modules loop
        read the record according to the
        module index of the hash table
        assign the module number, module
        index, number of callees field of
        the modules item with the values
        of the record
    for the number of callees loop
        assign the module index field for

```

```

        each called with the value of the
        record
    end loop
end loop
end

```

Procedure Display_Structure_Chart

```

begin
    read files to get number of modules,
    hash table and modules
    find recursively called modules
    do until user wants to exit
        loop
            Menu_and_Get_Choice
            case choice of
                1,2: to display structure chart
                    if choice = 1 then
                        Build_Up_Queue for the whole
                        system
                    else
                        Display_Module_List_and_
                        Get_Root_Module_Index
                        Build_Up_Queue starting from
                        the root module
                    end if
                    Draw_A_Tree
                    Draw_Module_Name_List
                    View_And_Move
                    enable to print the structure
                    chart

                3 : if the structure chart has
                    been drawn then print it

                4 : exit
            end case
        end loop
    end
end

```

V. CONCLUSION

The PCMS will help the system designer (or software engineers) to more effectively document and track software modules as they are being created. It is also a valuable tool to document, verify completeness and maintain an archivable tracking method for the product under review.

With the Pseudo_Code in each Module_Template, the source codes in Ada will be easily developed, and therefore, all system will be more easy to maintain.

REFERENCES

1. Fairly, Richard: Software Engineering Concepts, McGraw_Hill, 1985.
2. Cohen, Norman: Ada As Second Language, McGraw_Hill, 1986.

Dar-Hiau Liu is a professor at California State University, Long Beach, where he has been a faculty member since August, 1986. He teaches graduate and undergraduate classes in Software engineering, Distributed Computer System, Computing Theory and Programming Methods. His research interests include Software Reusability, Object Oriented Design, and Dynamic Task Scheduling in Distributed Computer System. Before coming to California State University, Long Beach he was a faculty member at Old Dominion University, Norfolk, VA. Previously, he was a Staff Engineer at IBM Corp. and was a project manager at IIT Corp. before that. He received his Ph.D. in Applied Mathematics and Computer Science from The University of Wisconsin-Madison in 1972. Previously, he had received a M.A. in Mathematics from Wayne State University, and a B.S. in Mathematics from National Taiwan Normal University. His current address is: Department of Computer Science and Engineering, California State University, Long Beach, CA. 90840.



DATA REDUCTION: AN ADA GENERICS METHOD

Warren D. Ferguson, Carol L. Carruthers,
Bruce J. Carter, Jr., Kenneth A. Staples, Jr., Charles F. Wise

General Electric Company
Government Electronic Systems Division
Morristown, NJ

Summary

This paper describes a data reduction method that automatically correlates data produced by Ada source code. The method employs a unique data reduction builder program that uses Ada generics to solve the configuration control problems associated with data reduction in many large systems. The presented method generates Ada generic instantiations to support the core functions needed by the data reduction analysis software to process recorded data. This method involves the recognition of recordable data structures by using identifiers embedded in the Ada source code. Thus an automated binding of the data structures occurs between the Ada source program and the data reduction analysis system.

Overview

A major problem in reducing the vast amount of data that a large scale system produces is an inability to automate the correlation of the output data with the source code that generates it. Data reduction software typically is written after all the data structures of the system have been developed, and the software runs as a separate program. A correlation between the two programs must exist regarding the definitions of the recorded data so that data analysis functions can be performed. In a development environment, source code often undergoes debugging and testing, and changes involving data structures and data outputs are routinely made to the source software. Often the corresponding changes to the data reduction software are inadvertently neglected. This can create a configuration control nightmare.

By enforcing a design discipline on the large-scale system that requires an Ada-type definition for each recordable data structure, this configuration problem can be eliminated. In addition to an Ada compiler and a host computer operating system, three major software components to be considered in describing this method are: (1) The Ada source software that produces recorded data, (2) the data reduction builder software, and (3) the data reduction analysis software that illustrates the relationship of one component to another.

Source Software for Recorded Data

As shown in Figure 1, the data reduction builder software accesses the Ada source software of the large scale system. The data reduction builder software identifies the recordable data structures and generates a symbol table that represents the data structures. By taking advantage of Ada's generics feature, the data reduction builder generates Type-Specific-Ada-Source-Code. This Type-Specific-Ada-Source-Code forms the core functions of the data reduction analysis software. The Ada compiler links the Type-Specific-Ada-Source-Code with the

source code that comprises the framework of the data reduction analysis software and produces an executable program. The executable program will reduce and analyze the recorded data produced by the large scale system. This automated source code extraction and expansion capability solves the configuration management problem of correlating output data with the source code.

Data Reduction Builder

The Ada source software for the large scale system that produces recorded data is processed by the data reduction builder in much the same way as a compiler processes source code. However, in the case of the data reduction builder, the only thing it is concerned with is to identify recordable data structures and to generate Type-Specific-Ada-Source-Code to process the data structures. For each Ada type used to define recorded data structures in the Ada source software for the large scale system, a procedure instantiation is generated with that type declared as the actual parameter to be associated with the generic formal parameter. The Type-Specific-Ada-Source-Code is placed into an Ada library and provides the core functions of the data reduction and analysis system. Figure 2 illustrates the major functions performed to generate the core functions of the data reduction and analysis system.

Data Reduction and Analysis System

The data reduction and analysis system provides an interactive user interface to analysis procedures that access recorded data contained in a data base. The Type-Specific-Ada-Source-Code

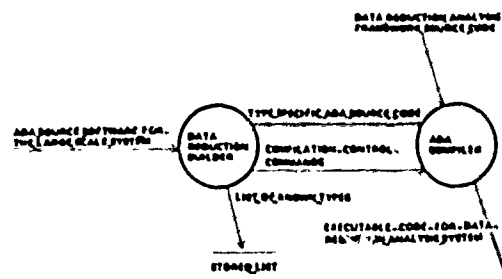


Figure 1: Data Reduction Support Environment

core functions support data base services used by the analysis procedures. The data base services handle functions such as the initial data base definition and loading of the raw recorded data. The raw recorded data is represented in the form of messages of various Known Types. Figure 3 illustrates the major processing functions involved. The software generated by the Data Reduction Builder mainly supports the Data Reduction Data Base Services.

Detailed Description

Ada-Type Definitions

For this method to be acceptable, the impact on the Ada source that produces recorded data must be minimal. Within the Ada source, a dedicated Ada-type definition is required for each recordable data structure. The method used to identify each recordable data structure selected by this design requires a key character string to prefix these Ada-type names. An alternate to the establishment of a prefix naming convention is to require the developers of recordable data structures to register the Ada-type names of these structures with the data reduction builder. The Ada-type name of each recordable data structure would be regis-

tered by placing it into an enumeration literal specification. The enumeration type that defines all recordable data structures would be used to guide the data reduction builder to determine the data structures to process. The drawback to this approach is that the certainty factor, that names in the enumeration type match the names of the type definitions for recordable data structures, is reduced because of the human activity involved. A tradeoff decision is to be made between the restrictions imposed by the use of a naming convention versus the added layer of automated configuration control gained by not using a manually maintained list of names.

Another requirement placed on the source code by the data reduction builder is that it must be placed in an Ada library, which infers that it has been compiled error-free. This relieves the data reduction builder of performing error checking that otherwise would be needed.

As input to the data reduction builder, the user must specify the name of the Ada program that produces recorded data and its program library. The user also may select compilation and link options.

The initial stage of Ada source code processing performed by the data reduction builder is to transfer all data structure definitions into a List.of.All.Types. This requires the recognition of the Ada data types, which are defined as the access type, array type, private type, record type, scalar type (which can be a discrete type or real type), and the subtype. Representation, specifications, and any use of the predefined language pragma PACK, associated with these types must also be extracted from the Ada source code.

Representative Specifications

It is assumed that the physical structure of a recorded message is controlled, either through the default structure provided by the Ada run time environment, through the use of Ada representation specifications, or through use of the predefined language pragma PACK. One important consideration is that when the system that produces the data uses different device drivers from the system that analyzes the data, a possibility for inconsistency exists. The use of Ada representation specifications in the Ada program that produces recorded data only affects the core functions that handle recorded data input to the data analysis software. The method used to handle a representation specification for a type is to copy it from the Ada program that produces the recorded data and place it in the type declarations for the data analysis procedures. The same processing is also true for use of the predefined language pragma PACK as it applies to recordable data structures.

Symbol Table Generator Function

The symbol table generator function involves creation of a List.of.Known.Types data structure that defines all of the type dependencies of each Known.Type. In this context, a Known.Type is the name given to the Ada type for an identified recordable data structure. An Applicable.Type is a type dependency that must be defined in order for the compilation of a Known.Type to be correct. Each Applicable.Type also gets defined in the List.of.Known.Types. Figure 4 illustrates the processing elements involved with creation of the List.of.Known.Types.

The List.of.Known.Types also contains links that represent the association of types in the Ada program that produces recorded data. Backwards links between a Known.Type and all of its Applicable.Types are generated. A backwards link from each Applicable.Type to all of its Applicable.Types is generated. Forward links from an Applicable.Type to all of the Known.Types and/or other Applicable.Types that refer to it are also generated. The purpose of generating all of these links is to support the Ada code generation stage of the data reduction builder and to support an interactive type analysis function.

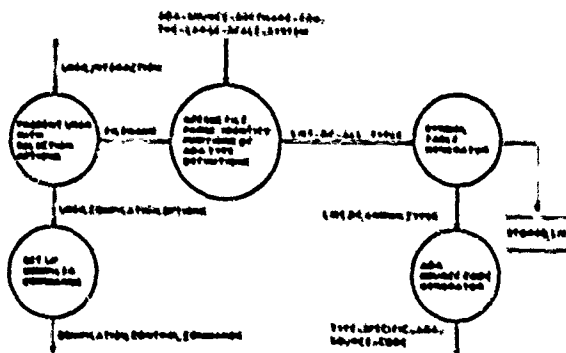


Figure 2. Overview of Data Reduction Builder

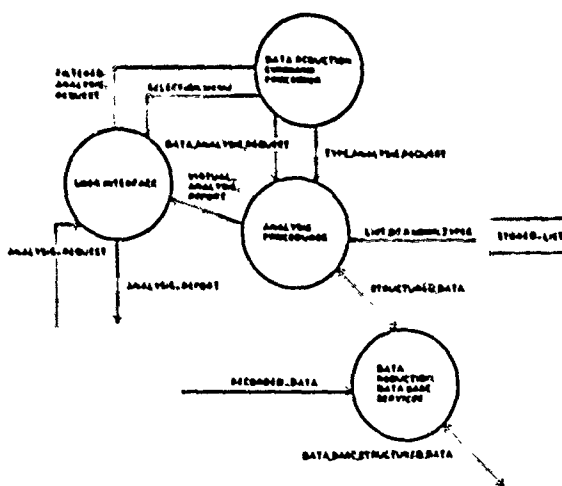


Figure 3. Overview of Data Reduction and Analysis System

The source code generator uses the List of Known Types to produce Ada code for procedure specifications, type definitions, and instantiations for Known Types.

1. Declare the Known, Type and Applicable Type data structures
2. Create a data base to contain recorded data
3. Transfer recorded data from the large scale system's storage media into the data base
4. Retrieve recorded data from the data base
5. Store recorded data into the data base

For each Known-Type, Ada code is also generated to declare instantiations of generic procedures, which are subsequently created by the compiler and become part of the library. Ada source code is also generated to create two enumerated types; one is the type Core Functions, which contains enumeration literals that identify each core function, the other is the type Known-Types, which identifies each of the Known-Type names that in turn, identify the recordable data structures. The types Core-Functions and Known-Types are used in combination by the data reduction and analysis procedures, to invoke the core functions on a type name basis. Source code is also generated to define each Known-Type and each Applicable-Type that gives viability of those structures to the data reduction and analysis software. As mentioned earlier, for recordable data structures (those identified

Analysis Procedures

Another analysis procedure that can be supported by the core functions is an interactive data type trace-forward trace-backward tool. This allows a user to examine source code data structure definitions through all levels of type, subtype, record, and array definitions, back to the primitive data structure elements. The trace-forward presents the user with all types that use an Applicable Type in their definition. The List of Known Types created by the data reduction builder is the key element for this feature, and serves a function similar to that of a data base schema.

Another interesting aspect of this method is that the data reduction builder can be used to generate Ada software to analyze its own data structures. This bootstrap approach to testing is convenient because it does not require the large scale system that produces recorded data to be completed before complex data structures can be used as test cases for the data reduction and analysis system. The use of Ada generics also has the obvious advantage that changes to the core functions or the inclusion of additional core functions only require changes to the generic bodies for primitive data types. Testing of the core functions also only needs to be performed on the primitive data types.

Using an Ada data structure, some features of an interactive recorded data examination tool will be presented. This tool makes use of the core functions that the Data Reduction Builder generates. Assume that a data structure has been recorded by a process that monitors communications bus activity in the large scale system. Also assume that a user has been able to select this data structure for analysis. The recorded data structure includes message header information containing the message identifier (which is its Ada type identifier), a timestamp, and possibly the sender and intended receiver of the message; the header is not defined as part of the example.



```

package sample_structure is
--
-- Data Structure Definition
--
type LAN_NODE_ID      is range 0 .. 1024 ;
type PERIPHERALS      is
( DISK_CAP ,
  SHARED_RAM ,
  DISPLAYS ,
  PRINTERS ) ;
type PROCESSORS       is ( MicroVAX, MacIntosh ) ;
type SYSTEMS          is ( A_SYSTEM, B_SYSTEM ) ;
type CONFIGURATION    is array ( PERIPHERALS ) of
  INTEGER range 0 .. 10_000 ;
type NETWORK          is array ( PROCESSORS ) of
  INTEGER range 0 .. 100 ;
type SYSTEM_RECORD    is record
  PRIMARY_ID , SECONDARY_ID : LAN_NODE_ID ;
  DEVICES : CONFIGURATION ;
  CPU_LEFT : NETWORK ;
end record ;
type SYSTEM_STATUS_MESSAGE is array ( SYSTEMS )
  of SYSTEM_RECORD ;
--
-- Data Structure Initialization
--
SYSTEM_ID : SYSTEM_STATUS_MESSAGE := (
  A_SYSTEM =>
    ( PRIMARY_ID      => 100 ,
      SECONDARY_ID    => 200 ,
      DEVICES =>
        ( DISK_CAP      => 1000 ,
          SHARED_RAM    => 900 ,
          NODES         => 200 ,
          PRINTERS      => 50 ) ,
      CPU_LEFT =>
        ( MicroVAX      => 50 ,
          MacIntosh     => 15 ) ) ,
  B_SYSTEM => ( 300, 400,
    ( 5000, 2000, 1000, 250 ) ,
    ( 45, 20 ) ) ) ;
end sample_structure ;

```

An integer dump of a message containing this data structure as initialized would be as follows:

```

0) 100
1) 200
2) 1000
3) 900
4) 200
5) 50
6) 50
7) 15
8) 300
9) 400
10) 5000
11) 2000
12) 1000
13) 250
14) 45
15) 20

```

At this point enough information is involved so that it is already difficult to determine what each number represents. The format of the message displayed in context with its Ada source code type definition is presented next. It includes an index from the beginning of the message as a reference. The index scheme could also reference word addresses into the message, bit fields within words, and multidimensional array positions. Arrays of records could also be presented in a way that makes them clear to understand.

```

SYSTEM_ID : SYSTEM_STATUS_MESSAGE :=
A_SYSTEM =>
  PRIMARY_ID => 100 ( 0 )
  SECONDARY_ID => 200 ( 1 )
  DEVICES =>
    DISK_CAP => 1000 ( 2 )
    SHARED_RAM => 900 ( 3 )
    NODES => 200 ( 4 )
    PRINTERS => 50 ( 5 )
  CPU_LEFT =>
    MicroVAX => 50 ( 6 )
    MacIntosh => 15 ( 7 )
B_SYSTEM =>
  PRIMARY_ID => 300 ( 8 )
  SECONDARY_ID => 400 ( 9 )
  DEVICES =>
    DISK_CAP => 5000 ( 10 )
    SHARED_RAM => 2000 ( 11 )
    NODES => 1000 ( 12 )
    PRINTERS => 250 ( 13 )
  CPU_LEFT =>
    MicroVAX => 45 ( 14 )
    MacIntosh => 20 ( 15 )
type SYSTEM_RECORD is record
  PRIMARY_ID, SECONDARY_ID : LAN_NODE_ID ;
  DEVICES : CONFIGURATION ;
  CPU_LEFT : NETWORK ;
end record ;

```

The user interface for this interactive recorded data examination tool will be similar to that of the MacIntosh, making use of a mouse, windows, and pull-down menus.

By using the mouse to select a recorded object, the user will be able to perform operations on the data, and will be able to investigate attributes of the object. One of the more useful features of the tool is the trace back of data types to their primitive type structure. If the user were to select the object "DEVICES" to be operated upon using a definition traceback operation, the following relevant information would be displayed:

```

type PERIPHERALS is
( DISK_CAP ,
  SHARED_RAM ,
  NODES ,
  PRINTERS ) ;
type CONFIGURATION is array ( PERIPHERALS ) of
  INTEGER range 0 .. 10_000 ;
type SYSTEM_RECORD is record
  PRIMARY_ID, SECONDARY_ID : LAN_NODE_ID ;
  DEVICES : CONFIGURATION ;
  CPU_LEFT : NETWORK ;
end record ;

```

It is assumed that, in most cases, a message data structure has the same definition in the sender and receiver. When that is not the case, a layer of decision making is involved where the user specifies this context.

Conclusion

This paper explains how the generics feature of the Ada language can be used to solve a data reduction configuration management problem that is typical of many large scale systems. What is unique about this method is the use of a tool to automatically generate the declarations of instantiations of the generic procedures, based on identification and decomposition of the recordable data structures.

Biographies



Warren D. Ferguson is currently the responsible engineer for the development of a data reduction analysis system supporting a submarine combat system at the Engineering Department of the General Electric Government Electronic Systems Division in Moorestown, New Jersey. Mr. Ferguson received a Masters of Science in Computer Science from Wright State University in Dayton, Ohio in 1981, and has worked as a software engineering consultant on many defense programs. He has been involved with all phases of development of real-time embedded systems, simulations, and test facilities, and has worked on several efforts involving Ada. He has a deep respect for the ideals of software commonality through the use of Ada, the application of the principals of artificial intelligence to computing systems, and a commitment to practicing a professional work ethic.



Carol L. Carruthers is involved in the development of a data reduction analysis system supporting a submarine combat system at the Engineering Department of the General Electric Government Electronic Systems Division in Moorestown, New Jersey. Ms. Carruthers received a Bachelor of Science degree in Management Information Systems from Glassboro State College in Glassboro, New Jersey in 1983, and is currently completing a Bachelor of Science degree in Computer Science at Glassboro State College. Her experience as a software engineer has been primarily in user interfaces.



Bruce J. Carter, Jr. is a software engineer with over ten years of data reduction experience supporting the testing and evaluation of large military systems. From 1979 to 1986, he served as a Computer Systems Officer in the USAF. His work experience involves a variety of areas including radar system development and tests, surface-to-air missile simulations, electronic warfare range instrumentation, war gaming exercises, real time operational flight program controls and tests, analog and digital computer programming, and cruise missile test and evaluation. Mr.

Carter earned a Bachelor of Science degree in Mathematics/Physics from Brigham Young University in 1979. He currently serves as a consultant, designing a data reduction analysis system for the submarine combat system under development by the Engineering Department of the General Electric Government Electronic Systems Division in Moorestown, New Jersey.



Kenneth A. Staples, Jr. is currently participating in the development of a system for supporting the creation and maintenance of the tactical data base for a submarine combat system at the Engineering Department of the General Electric Government Electronic Systems Division in Moorestown, New Jersey. Mr. Staples received a Bachelor's of Science degree in Aerospace Engineering from the University of Virginia in 1982, and has been designing and developing systems using Ada for the past five years.



Charles F. Wise is manager of the Software Development Facility in the Engineering Department of the General Electric Government Electronic Systems Division in Moorestown, New Jersey. Mr. Wise received a BS in Computer Science from Temple University and has authored several papers on automated software development environments. He is responsible for large scale computer facilities for engineering and development in both hardware and software at a corporate level. His experience as a computer professional dates from 1967 and includes the design and development of operating systems, management of real-time communications system development, and the design of micro-processor based real-time systems.

A METHOD OF TRANSLATING FUNCTIONAL REQUIREMENTS FOR OBJECT-ORIENTED DESIGN

Russell Brown
Verlynda Dobbs

Wright State University

ABSTRACT

The use of Object-Oriented Design methods for DoD systems developed in Ada presents a number of challenges. One of the most critical is the difficulty of maintaining traceability between functional requirements and an object-oriented design. This paper presents a forms-based methodology called Functional Requirements Translation (FRT), which can be used as a framework for translating functional specifications into a set of object requirements. Each requirement is documented on a Requirement Translation Sheet (RTS) and is translated to a set of objects, operations, and access links between objects, which are necessary to satisfy the requirement. These object requirements are then combined from all of the RTSs onto Object Requirement Sheets (ORSs). Each ORS documents the operations and access links of a single object and contains references back to the individual RTSs which generated them. This method provides traceability in both directions of the translation and can be used to identify unsatisfied requirements and produce good detailed object designs.

1. INTRODUCTION

The U.S. Department of Defense (DoD) has developed and promoted the Ada programming language in an attempt to lessen the impact of the software crisis being brought on by the increasing complexity of software systems. Ada enforces, or at least supports, basic principles of good software engineering such as *abstraction*, *information hiding*, and *modularity*. These principles are brought together nicely in a design methodology called Object-Oriented Design (OOD), proposed by Booch[4]. In OOD, software systems are decomposed based upon abstract representations of objects in the problem space. This differs a great deal from more

common approaches of decomposing systems into functions. For a basic introduction to the principles of OOD, the reader is referred to [3].

But OOD, as presented by Booch, is only a design and implementation approach. The first edition of his book does present an informal requirements analysis approach based upon the work of Abbot [1], but he never claims that it should take the place of a formal requirements analysis. To avoid further confusion, it was left out of the second edition[5]. So basic OOD provides no support for requirements analysis or maintenance of requirements traceability to the design.

Because of this, Booch and other researchers have suggested the use of established analysis methods such as Structured Analysis[9], JSD[13,16], and VDM[12]. These methods can be very useful for clarifying system requirements and developing structured requirement documents, but they do not completely solve the problem of requirements traceability. Data flow analysis, for example, produces a structured set of functional requirements, each of which may be satisfied by a complex set of objects, operations, and object visibility. The difficulties of using data flows with OOD have been great enough for at least one DoD contractor to abandon OOD completely[6]. The problem is less pronounced with VDM and JSD, since the requirements they produce contain more information about the behavior of problem-space objects.

The difficulties of maintaining requirements traceability with OOD are compounded by the way the DoD does business. The development specifications for DoD systems are functional and are not usually subjected to any formal requirements analysis techniques. Some software developers have applied Structured Analysis to formalize specifications before proceeding with design, but ultimately the system must be tested for compliance with the original requirements.

If DoD software developers are to take maximum advantage of the software engineering features of Ada, some method of maintaining functional requirement traceability to OOD must be developed. This paper presents a structured, forms-based methodology for translating functional system requirements into object requirements, while maintaining traceability between them. It is flexible enough to work with many different types of requirements, and it does not enforce any particular design strategy (other than object-oriented decomposition). It is specifically intended for use on DoD systems implemented in Ada.

2. THE TRACEABILITY GAP

We can see that there is a gap between requirements analysis methods which produce functionally decomposed specifications, and OOD, which seeks to develop a design decomposed into objects. Some software designers have found that the simplest way to deal with this problem is to review the functional specifications, then press on with OOD and hope all of the requirements are satisfied by the final design. Another approach is to assume that each function called out in the specification corresponds to an operation on some object; the concept of an object is used like glue to hold a group of related functions together. Using this approach, it is difficult to produce a design which models the real world problem. More formal approaches, such as SEL's Abstraction Analysis[18], have been suggested, but each seems to assume that there should be a simple method of converting functional requirements into object-oriented designs, similar to the methods used to convert structured data-flow diagrams into structured designs. These approaches ignore the fact that a good object-oriented design requires information about the real problem which may not exist in the functional requirements. The translation of functional requirements must remain a creative human task; when using OOD, the software developer must design with the real problem in mind, choosing objects which best model the problem space.

3. OBJECT REQUIREMENTS

A method is needed for maintaining requirements traceability between functional requirements and OOD. The methodology presented in this paper is based upon the translation of functional requirements into object requirements which can then be easily traced to the design. The information which must appear in an object requirement is the object name, state description, and definitions of operations and access links. In addition, each object is categorized as either an abstract state machine (ASM), or an abstract data type (ADT)[5].

Object names are text strings which uniquely identify objects. They are used throughout the methodology to reference objects. Legal Ada identifiers should be used to name objects to make implementation easier.

An object's state description provides an English description of what the object is meant to represent and its range of possible states. Only the state information which is in some way accessible to other objects through operations should be presented.

The operations on an object are specified formally with an operation name, parameter objects, and returned objects. All parameters are assumed to be passed as values to the operation, so all information provided by the operation is in the returned objects (implemented in Ada as either values of functions, access types, or 'out' parameters). An English description of each operation can also be provided, although none are used in this paper. The format used to specify operations looks like this:

Oper_name(Par_1:Type, Par_2:Type) => Return_1:Type.

Each object requirement must also include the names of other objects which it must access to define its state and perform its operations. These access links represent visibility from one object to all of the operations of another object. Other researchers have suggested that the links between objects should include a declaration of which specific operations are accessed[2]. Some have even suggested that each operation should have a separate set of links to the operations on other objects which it accesses[18]. Although these methods may provide a more detailed representation of control within the object structure, there is no easy way to implement such limited visibility in Ada. Since FRT is specifically intended to support OOD with Ada, a simple object-to-object link is used. This is also the approach used by Booch[3,5] and Buhr[8].

An object requirement for a television set (apparently with automatic color and contrast adjustment) is shown below. The operations require no explanation; based upon the state description, the purpose of each is obvious. To provide a picture, the television must receive a signal, so this is not a complete model. However, it does show the basic format of an object requirement.

Object Name:	Television_Set
State Description:	The state of a Television_Set consists of the channel it is tuned to, and it's on/off state.
Operations:	Turn_On. Turn_Off. Change_Channel(To_Channel:Channel). Get_Current_Channel=> Current_Channel:Channel.
Access Links:	Picture_Tube Tuner Electrical_Outlet Channel

4. CAPABILITIES OF METHODOLOGY

A formal methodology for bridging the traceability gap should:

- Support Generation of object requirements from functional requirements
- Provide Identification of unsatisfied functional requirements
- Provide Identification of unnecessary object requirements
- Be as automated as possible.

The methodology must support the requirements analyst while object requirements are being developed. As stated previously, this does not mean it should automatically generate the objects, but it should provide a structured approach. The methodology should be useful with both functional English text requirements and products of structured requirements analysis techniques[9].

If all of the objects and operations necessary to satisfy a functional requirement exist in the current set of object requirements, we can consider it to be *satisfied*. The methodology must provide the capability to easily identify those functional requirements which are not satisfied in this context. This capability must be available at any point during the translation of functional requirements to object requirements.

If, for some reason, object requirements have been generated which are not traceable to a functional requirement, the methodology must provide the capability to identify them as *unnecessary*. This capability becomes more important when the design must change in response to changes in the system requirements.

In their final report for the Ada Simulator Validation Program[20], Burtel, Inc. points out the complex relationships which can exist between functional requirements and objects in an object-oriented design. They go on to suggest that "Manual methods of tracking requirements are insufficient for handling the large number of requirements and objects needed for a simulator." Since we would also like a method which can scale up for use on large software projects², we should heed this advice and identify a methodology which can be automated. The methodology presented in this paper is being applied manually to prove its concept, but in the future it will be supported by an automated tool.

5. FUNCTIONAL REQUIREMENTS TRANSLATION

The methodology we are proposing is Functional Requirements Translation (FRT). It provides a flexible approach for maintaining requirements traceability while an object-oriented design is being developed from functional specifications. FRT will be presented here as a manual method which meets three of the requirements listed previously. The fourth requirement, that the methodology be automated, will be met by developing a tool to provide the necessary record keeping and reports. When using FRT, a requirements analyst, with knowledge about the problem space and the object requirements already defined, can translate each individual functional requirement into a set of objects, operations, and access links which will satisfy it. These object requirements are combined incrementally to produce the set of objects needed to satisfy all system requirements. In the process, traceability to the functional requirements is maintained for each individual object, operation, and access link.

² Flight simulators are typically 100,000 to 500,000 lines of code

Functional Requirements Translation is designed to track complex relationships between functional requirements, performance requirements, objects, operations, access links, and derived object requirements. It is built on the recognition that a requirements analyst must perform a mental transformation of the functional requirements back to the problem space in order to generate a good object-oriented design. It will also support simplified approaches to translating the requirements, such as a one-to-one mapping of functional requirements to operations.

Two forms are used in FRT: the Requirement Translation Sheet (RTS), and the Object Requirement Sheet (ORS). The completed RTSs and ORSs are the main products of the methodology, but others, such as complete object diagrams, a Master Requirements List, and a Master Objects List can be generated. Each RTS documents one functional or performance requirement and its translation to objects, operations, and access links. Each ORS documents a single object, its operations, access links, and the functional and performance requirements to which each is traceable.

An RTS consists of a header, a body, and an optional object diagram. A completed RTS is shown in Figure 1. The header contains a title for the requirement, an alpha-numeric identification, the requirement type (either *functional* or *performance*), a description which could just be the actual text of the requirement, and the status of the requirement. The ID number is a unique identifier for the requirement, either taken directly from a specification paragraph number or the numbering of nodes in a data flow diagram. The ID number should also indicate the requirement's source document (specification, change proposal, correspondence). The status of a requirement should be *untranslated*, *translated*, or *combined*. The meaning of these will become apparent as the FRT process is explained. The body of the RTS contains one entry for each object needed to satisfy the requirement, and each object entry contains definitions for the specific operations which are needed, and a list of other objects which must be accessed. Each RTS can also contain an object diagram, which is a graphic representation of the object entries. Only the objects, operations, and access links necessary to satisfy the requirement should appear on it. Except for some minor differences,³ the object diagrams in this paper follow the conventions established by the Goddard Space Flight Center, Software Engineering Laboratory[18].

³ SEL uses the term *Provides* instead of *Operations*, and *Uses* instead of *Access*. SEL *Uses* entries also list the specific operations used, not just the object.

REQUIREMENT TRANSLATION SHEET

HEADER

TITLE: Client Information on Reports
 ID NUMBER: spec.3.2.2
 TYPE: Functional
 DESCRIPTION: The client's name and address must appear
 at the top of all reports.
 STATUS: Translated

BODY

OBJECT: General_Journal_Report
 ACCESS LINK: Client
 ACCESS LINK: Person_Name
 ACCESS LINK: Address

OBJECT: General_Ledger_Report
 ACCESS LINK: Client
 ACCESS LINK: Person_Name
 ACCESS LINK: Address

OBJECT: Client
 OPERATION: Name(Of_Client : Client) => Client_Name : Person_Name.
 OPERATION: Address(Of_Client : Client) => Client_Address : Address.
 ACCESS LINK: Person_Name
 ACCESS LINK: Address

OBJECT: Person_Name
 OPERATION: Print_on_Report(The_Name : Person_Name).

OBJECT: Address
 OPERATION: Print_on_Report(The_Name : Person_Name).

OBJECT DIAGRAM

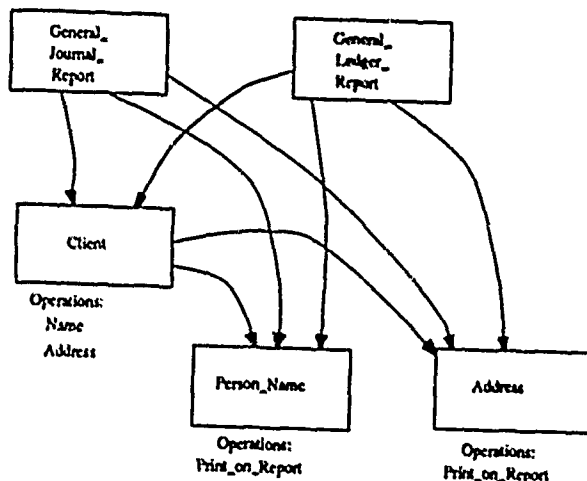


Figure 1. Completed RTS

A completed ORS is shown in Figure 2. It consists of a header, a body, and an optional object diagram. The ORS header contains the object name, type (ASM or ADT), level, and state description. These have all been described already, with the exception of object levels, which will be explained later. The ORS body contains one entry for each operation or access link the object possesses, and each entry contains the identification number of the functional and performance requirements to which the entry item (operation or access link) can be traced. Operation entries must define the operation's parameters and returned values, and access link entries must name the accessed object. Entries which represent derived operations or access link requirements should reference the objects which use them. ORSs can also contain an object diagram which represents the object, its operations, and any access links to other objects.

OBJECT REQUIREMENT SHEET

HEADER

NAME: Client
 TYPE: ADT
 LEVEL: unknown
 STATE DESCRIPTION: Client is an abstract data type which manages client information. It contains the client's name, address, and fiscal year structure.

BODY

OPERATION: Name(Of_Client : Client) => Client_Name : Person_Name.
 FUNCTIONAL: spec.3.2.2

OPERATION: Address(Of_Client : Client) => Client_Address : Address.
 FUNCTIONAL: spec.3.2.2

OPERATION: Establish_Fiscal_Year(The_Client : Client).
 FUNCTIONAL: spec.3.1.4, corr.12.3

ACCESS LINK: Person_Name
 FUNCTIONAL: spec.3.2.2

ACCESS LINK: Address
 FUNCTIONAL: spec.3.2.2

ACCESS LINK: Fiscal_Year
 FUNCTIONAL: spec.3.1.4

ACCESS LINK: Calendar_Month
 FUNCTIONAL: spec.3.1.4
 PERFORMANCE: spec.5.2

OBJECT DIAGRAM

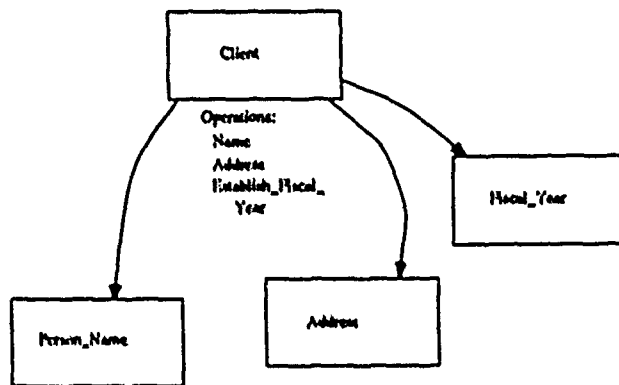


Figure 2. Completed ORS

There are six steps in the FRT process:

1. Classify Requirements
2. Identify Preliminary Objects
3. Translate Functional Requirements
4. Combine Object Requirements
5. Identify Derived Requirements
6. Establish Traceability to Performance Requirements

They are not necessarily sequential, as can be seen on the data-flow diagram in Figure 3.

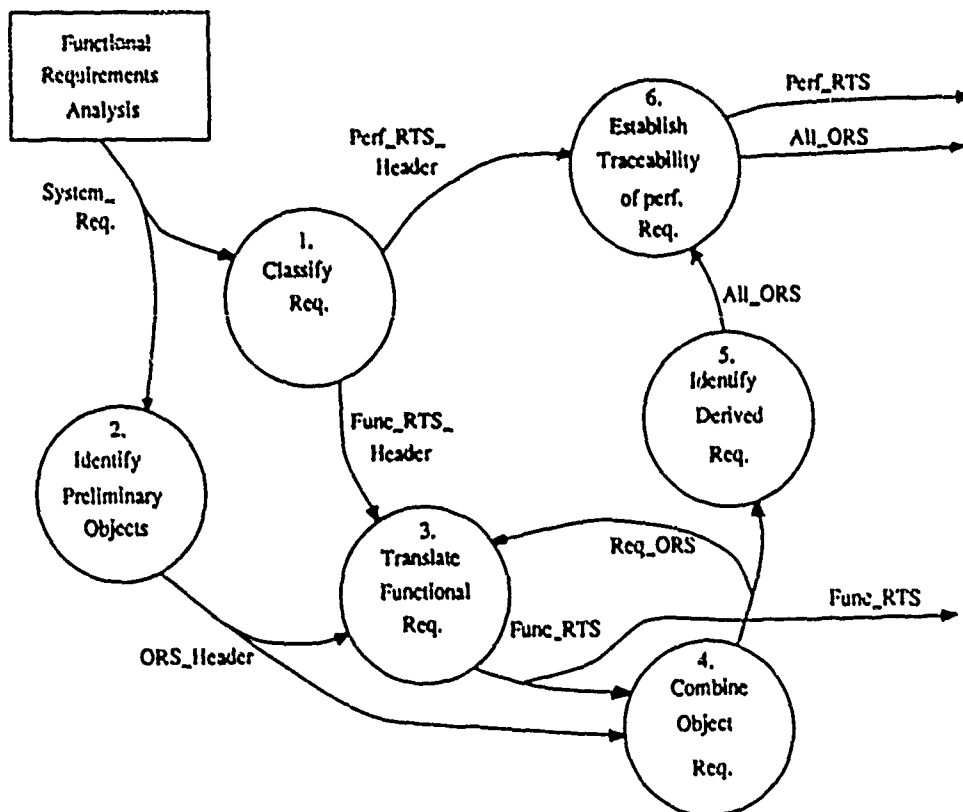


Figure 3. The Functional Requirements Translation Process

5.1. CLASSIFY REQUIREMENTS

The purpose of this step is to organize system requirements and classify them as *functional* or *performance*. The requirements must be classified because functional and performance requirements are translated differently, during separate steps of the FRT process. During the Classify Requirements step, an RTS is created for each system requirement and the requirement ID number structure is established. It may also be desirable to break a large system into subsystems if simple interfaces exist. If this is done, the requirements should be divided accordingly and FRT should be applied separately to each subsystem. No differentiation is made between hardware and software requirements during the FRT process; this is left as a design task.

A Requirements Translation Sheet is generated for each requirement, but only the following header information is filled in:

Title
ID Number
Type (Functional or Performance)
Description/Text
Status

The body of the RTS will be filled out when the requirement is translated into object requirements. For easy reference, RTSs should be filed by their ID numbers. It may also be useful to keep a Master Requirements List showing the ID number, title, type, and status of each. Every requirement should begin with a status of *untranslated*.

5.2. IDENTIFY PRELIMINARY OBJECTS

During this step, informal methods are used to define a starting set of objects, operations, and access links. This establishes a top-level structure for the object requirements and anchors representations of problem-space objects which may not be clearly represented in the functional requirements. This step can also provide a common vocabulary for a group of analysts working on the same translation; identifying a preliminary set of objects may help prevent the creation of multiple objects with different names which represent the same problem-space entity, or multiple ASM objects which could have been a single ADT object.

A number of methods for identifying potential objects have been suggested in the OOD literature. Although Grady Booch no longer supports the Informal Description as a formal design method, this is one place where it could be useful. Top-level descriptions of the system would probably be good sources of preliminary problem-space objects, and the names given to items in a data-flow diagram could also be a good source.

Each preliminary object is established by filling in at least the object name, type (ASM or ADT), and a partial state description on an Object Requirement Sheet. If possible, the ORS header can be filled in completely and a set of

operations and access relationships can be defined in the body of the ORS. If a Master Objects List is used, the name and type of each preliminary object should be added to it. Until each object, operation, or access link is assigned traceability back to a functional requirement or another object, it is considered *unnecessary*. Any preliminary objects which are still unnecessary after translation of all functional requirements and identification of all derived objects should be examined as candidates for deletion.

5.3. TRANSLATE FUNCTIONAL REQUIREMENTS

An analyst translates a functional requirement into object requirements by identifying a set of objects, operations, and access links which will satisfy it. If possible, these objects should be taken from the basic set of objects already established on ORSs. An existing object may not have the exact operation needed or an access link to a certain object, but identifying these during translation will ensure they become part of the object requirement in the future. An example requirement for an accounting system is shown below, along with the objects, operations, and access rights to which it might translate.

REQUIREMENT

The system shall allow the user to establish a different fiscal year starting month for each client.

TRANSLATION

OBJECT: Accountant
ACCESS LINK: Client

OBJECT: Client
OPERATION: Establish_Fiscal_Year(The_Client:Client)
=> The_Client:Client.
ACCESS LINK: Fiscal_Year
ACCESS LINK: Calendar_Month

OBJECT: Fiscal_Year
OPERATION: Create => New_Fiscal_Year : Fiscal_Year.
OPERATION: Set_First_Month(The_Fiscal_Year : Fiscal_Year,
First_Month : Calendar_Month)
=> The_Fiscal_Year : Fiscal_Year.
ACCESS LINK: Calendar_Month

OBJECT: Calendar_Month

Again, both Grady Booch's Informal Description method and the terminology used in a data-flow diagram can help identify the needed objects. But, existing objects should be checked before generating new ones.

Required objects are documented on the Requirement Translation Sheet by making an entry for each in the body. An object diagram showing these objects can also be added. Each RTS entry should contain the definition of one object's operations and a list of its access links to other objects. After all necessary entries have been made, the status of the RTS should be changed to *translated*.

It is not necessary to define the operations of the lowest level objects (the ones which do not access others). In fact, if their operations were defined, they would require access to the objects they pass as parameters or return. This would require every translation to define objects and operations down to the most basic objects already defined by the implementation language (Integer, float, duration for Ada). In the example above, Calendar_Month could have defined operations such as:

OPERATION: Create(The_Month:Month_Number)
=> New_Month:Calendar_Month.

But this would require another object, Month_Number, which is accessed by Calendar_Month. To keep translations simple, they should contain only those objects which directly relate to the requirement. Lower level objects which just support the translation should be defined later as derived requirements.

5.4. COMBINE OBJECT REQUIREMENTS

During this step of FRT, the translations on the RTSs are combined to form object requirements. Each object requirement on an RTS is used to generate a new Object Requirement Sheet (or update an existing one, and traceability to the functional requirement is maintained. This step should occur whenever a group of completed RTSs is available. This will make any new objects immediately available to the analysts doing the functional translations.

Each object requirement on an RTS must be transferred to the ORS for that object. If no ORS exists, one must be created by filling in the necessary header information. A separate entry is made in the ORS body for each operation or access link listed for that object on the RTS. The ID number from the RTS is then added to each of these entries so that each operation and access link can be traced separately back to it. If an ORS entry already exists for an operation or access link, the functional requirement ID number must still be added. This means one operation or access link can be traceable to more than one functional requirement. After transferring the object information from the RTS to the ORSs, the new objects should be added to the Master Objects List (if one is used), and the status of the RTS should be changed to *combined*.

After updating the ORSs for each of the object entries on the RTS, the analyst should check for access loops. Each new access link which was added to the set of objects could potentially have completed a loop of access links. A path of access links passing from an added or updated object,

through other objects and their access links, back to the starting object, cannot be directly implemented in Ada. When a loop is found, it must be broken by changing at least one requirement translation. One possible solution is the addition of a 'communication' object as shown in Figure 4. If object_A and object_B must send messages to each other, they should not access each other - both should access a separate message object, and if possible, the object used to break the loop should represent a problem space object. The methodology presented in this paper is currently manual, so the recommended method of searching for these loops is inspection by the analyst; no formal algorithm is presented.

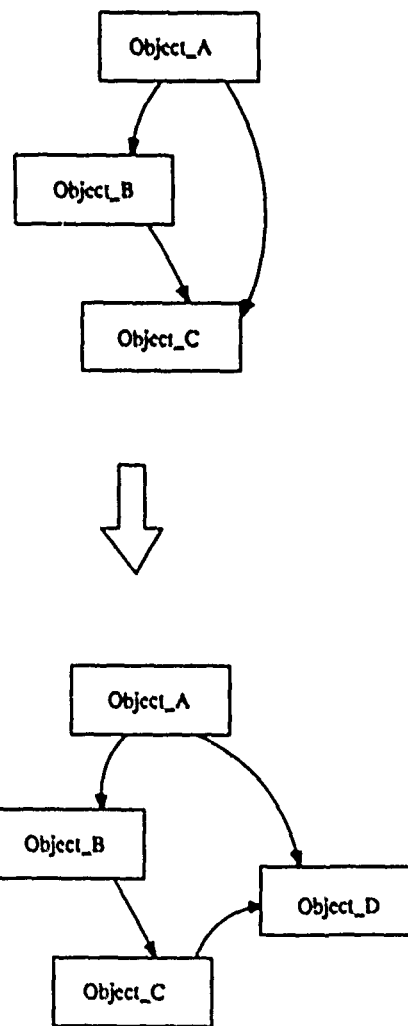


Figure 4. Breaking Access Cycles

Object level numbers should be updated periodically during the process of combining RTSs. The level number of any object which is not accessed by any other object is zero. The level of any other object is one deeper than the deepest level object which accesses it (greater numbers represent deeper levels). Figure 5 shows the access structure of a group

of objects and their resulting level numbers. Level numbers provide a clear hierarchical structure and can be used to define subsets of objects such as "all objects down to level 3" or "Object_A and all objects it accesses down to level 5". It is probably not a good idea to update level numbers after each RTS is added, because a change in the way one object is accessed may change the level number of many other objects. At the very least, however, the level numbers of objects should be calculated after the last RTS is combined.

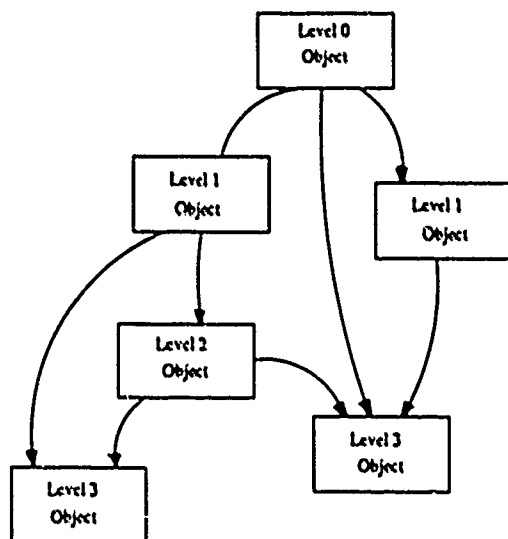


Figure 5. Level Number Example

5.5. IDENTIFY DERIVED REQUIREMENTS

The set of object requirements is not yet complete when all of the system requirements have been translated. Additional objects and operations will have to be added to support the existing objects; these are derived object requirements. Derived objects may have to be added for several reasons. First, the translation may not have produced all of the top level objects necessary to provide a clear hierarchical structure for the system.⁴ Second, new operations and access links may have to be added to existing objects. It is certainly not reasonable to expect an analyst to identify all needed operations during the initial translation. Many of these requirements will not be apparent until detailed design of individual

⁴ A good indication of this is the presence of more than one level zero object.

operations are produced. Finally, additional low-level objects may be needed to support the existing objects. If possible, these objects should represent deeper levels of the problem space, but at some point it may be necessary to define purely solution space objects such as generic data structures.

A new ORS should be created for each derived object requirement (unless it was already created during the identification of preliminary objects or combination process). All of the header information should be filled in and entries made in the ORS body for each of the derived object's operations and access links. The derived requirement entries are completed by adding the name of the existing objects which require the existence of the operation or access link. This means derived object requirements are only traceable to other objects; a derived requirement becomes *unnecessary* when it cannot be traced to an existing *necessary* object. Each derived object should also be added to the Master Objects List.

5.6. ESTABLISH TRACEABILITY TO PERFORMANCE REQUIREMENTS

During this step, the requirements analyst determines the scope of each performance requirement and updates its RTS and the associated ORSs. Performance requirements are translated in much the same way as functional requirements, using the same forms. The objects, operations, and access links subject to the constraints of the performance requirement are entered on its RTS, and the performance requirement ID number is added to their ORS entries. The performance requirement ID numbers in an ORS entry should be set apart from the functional requirement ID numbers because they have different effects on the traceability of the object. For example, traceability to a performance requirement alone does not mean an operation or access link is *necessary*. Performance requirements do not generate object requirements, they just constrain their behavior.

6. APPLYING THE METHODOLOGY

FRT has the flexibility to support many different approaches to OOD. We can look at two simple approaches which have been used, then recommend an approach which combines the strong points of both. The recommended approach is more complex, but the added complexity can be handled by using FRT, and it can generate a better object-oriented design.

One simple form of OOD involves the separation of requirements traceability and design. The system designers look over the system requirements and then begin identifying objects and performing the design. No requirements traceability information is maintained. The fully developed system is then tested against the original requirements and the designers pray that it meets them. Obviously this is an extreme approach and is unacceptable for all but the simplest pieces of software. However, if a developer chooses to use it, FRT can be of some help. If the Identify Preliminary

Objects step is extended until the entire design is complete, FRT effectively becomes this approach. Designers who happen to know why they are defining a given object can easily take the time to make some entries on the RTSs and ORSs. Not all objects will be traceable, and not all requirements will appear *satisfied*, but some traceability information is better than none.

Another simple OOD method assumes a direct one-to-one relationship between functional requirements and operations on objects. Most applications of this approach have involved some front-end requirements analysis using data flow methods. Function nodes are translated directly to operations, and objects are identified to represent data stores and data flows and to *glue* related operations together. This design method provides straightforward requirements traceability, but can only produce a design which models the problem space if all of the problem space objects are present in the data flows. Again, FRT can support this approach if a software developer chooses to use it. The entire FRT process would be followed, but requirement translations would only consist of a main object, one operation on it, and whatever objects it had to access to perform that single operation.

We recommend an approach to FRT which can provide requirements traceability without sacrificing a good representation of the problem space. After classifying all of the system requirements, they should be reviewed and some emphasis should be placed on establishing a good top-level structure of preliminary objects which represent the problem space. There is no real danger in generating extra objects; if they really are unnecessary, FRT will eventually show it. Next, requirements should be translated one at a time, or in small groups, then combined as soon as possible with the existing ORSs. Analysts should feel free to identify preliminary or derived object requirements *at any time* during the FRT process.

A Master Requirements List and Master Objects List are probably not necessary on relatively small projects where it is practical to just flip through RTSs and ORSs. If these lists are used, it must be understood that they are just a summary of the information on the main forms. They should be checked periodically to make sure they are consistent.

Object diagrams should be used if the tools needed to maintain them are available. They provide a quick view of the object structure, and are generally easier to analyze than the written object requirements. FRT does not prescribe any format for a complete system object diagram, but it is obvious that it will have to be a hierarchy of separate diagrams. It is important to insure that the object diagrams match the information on the ORSs.

7. USING THE PRODUCTS

The primary products of Functional Requirements Translation are the Requirement Translation Sheets and the Object Requirement Sheets. Other, optional products are the Master Requirements List, Master Objects List, and the combined object diagrams, but these are just alternate representations of

the data on the RTSs and ORSs. These products can have many different uses, but they are mainly intended to support the design phase, identify unsatisfied requirements, and identify unnecessary object requirements.

FRT only supports the translation of functional requirements to object requirements. The classification of object requirements as either hardware or software requirements (or both) is left as a design issue. Also, FRT does not support Ada implementation decisions. Each ORS can be implemented as an Ada package, task, generic, or just a derived type. Operations can be procedures, functions, task entries, or even declarations. Tracing these Ada constructs back to their ORS should be straightforward (especially if the same names are used) and the ORS provides the traceability back to the original requirements.

The most important use for FRT products, from the developer's point of view, should be tracing object requirements back to the functional and performance requirements they must satisfy. This becomes a significant capability during the design phase, allowing the designer to see exactly what an object, or a set of objects, must accomplish.

The generated RTSs can be used to identify *unsatisfied* requirements. Any system requirement which has no RTS, or has an RTS with a status other than *combined*, is considered unsatisfied. The translation of the RTS may be complete, but if it has not been combined, the current object structure may not contain all of the objects, operations, and access links needed to satisfy it.

The ORSs can be used to identify *unnecessary* objects, operations, and access links. These have no valid reason to appear in the system design, based upon the functional requirements which have already been translated. An operation or access link can be considered unnecessary if its ORS shows no traceability back to a functional requirement or another object requirement. Any operation or access link which is only traceable to other unnecessary objects is also considered unnecessary. Objects are unnecessary only if they have no *necessary* operations or access links.

8. HANDLING CHANGING REQUIREMENTS

Besides helping verify that a design meets the requirements, FRT can also help analysts and designers deal with changes in the requirements. Most DoD system specifications change frequently, so we must be able to update our set of objects in response to new, deleted, or modified functional or performance requirements.

If a new requirement is added, it must be subjected to the entire FRT process. The requirement must be classified and documented on an RTS. If it is a performance requirement its traceability to existing ORSs must be established. If the requirement is functional, it must be translated and its translation must be combined with existing ORSs (possibly generating new ones). All performance requirements must be re-evaluated to see if the operations and access links generated by the new functional requirement are subject to their

constraints. All RTSs and ORSs must be updated accordingly.

If a system requirement is deleted, its RTS is destroyed and all references to it on ORSs are removed. Note that the deletion of a functional requirement could cause some objects to become unnecessary.

The most straightforward way to handle a modified requirement is to assume it was deleted completely, then added under the same ID number. It is also possible to just review the existing RTS for the requirement and make any necessary changes to its translation and effected ORSs.

9. FUTURE IMPROVEMENTS

There are many ways FRT could be expanded and improved in the future. It is currently a manual method, and could be very cumbersome for large systems. It also provides very little direct support for the design phase. Besides these two major areas, there are many other small improvements which could help make FRT more acceptable to developers of Ada systems.

The most important future improvement in FRT is its automation. This will be accomplished by developing a tool for the creation and maintenance of the RTSs and ORSs. Such a system could easily generate Master Requirements Lists and Master Objects Lists, and could identify unsatisfied requirements and unnecessary objects. Other features which could be useful are the automatic combination of completed RTSs with existing ORSs and identification of access loops in the object structure.

FRT could also be expanded to better support the design phase. It should allow the analyst or designer to differentiate between hardware and software requirements, and it should record decisions about the Ada constructs which will be used to implement each object and operation. It would also be useful to extend the object requirements to include definitions of the exceptions handled and generated by each object.

There are many other improvements which could make FRT more comprehensive and useful. For complex systems which exist in many configurations, it would be helpful to have some sort of version numbering facility. Such a system could also benefit from a better method of tracking the source of individual requirements (currently this information is contained in the ID number). A more complex improvement, but one which has been accomplished on other OOD support tools[2], is integration with a drawing tool which can be modified to support OOD and provide parsable output to FRT.

10. CONCLUSIONS

We have introduced Functional Requirements Translation, a methodology for translating functional requirements to object requirements while maintaining requirements traceability. It

is intended for use with OOD and Ada on DoD systems. FRT is flexible and relatively simple to use, and it can support most requirements analysis methods and object identification methods being used with Ada. It is simple because it is based around only two forms and involves no complex algorithms. It allows the requirements analyst to perform the translation with confidence that all of the system requirements are being addressed.

Researchers have addressed many different aspects of applying OOD with Ada, but very few have taken a practical approach to maintaining requirements traceability. In the future, it is possible that FRT, with the support of an automated tool and some enhancements to support the design phase, will help bridge this traceability gap.

REFERENCES

1. Abbot, R.J., "Program Design by Informal English Descriptions," *Comm. ACM*, Vol 26, No 11, Nov 1983.
2. Ballin, S.C., "An Automated Quality Assessor for Ada Object-Oriented Designs," *Proceedings of National Aerospace and Electronics Conference*, IEEE, May 1988.
3. Booch, G., "Object Oriented Development," *IEEE Trans. Software Eng.*, Vol SE-12, Feb 1986.
4. Booch, G., *Software Engineering with Ada*, Benjamin/Cummings, 1983.
5. Booch, G., *Software Engineering with Ada*, 2nd Ed., Benjamin/Cummings, 1983.
6. Brown, R.J., "Requirements Analysis for OOD," Wright State University, Winter 1988.
7. Brown, R.J., "Developing Object-Oriented Designs from Functional Requirements: Two Case Studies," Wright State University, Spring 1988.
8. Buhr, R.J.A., *System Design with Ada*, Prentice-Hall, 1984.
9. DeMarco, T., *Structured Analysis and System Specification*, Prentice-Hall, 1979.
10. Fairley, R.E., *Software Engineering Concepts*, McGraw-Hill, 1985.
11. General Electric Company, *Software Engineering Handbook*, McGraw-Hill, 1986.
12. Jackson, M.I., "Developing Ada Programs Using the Vienna Development Method (VDM)," *Softw. Pract. and Exper.*, Mar 1985.

13. Jackson, M., *System Development*, Prentice-Hall, 1983.
14. Kachler, T. and D. Patterson, "A Small Taste of Smalltalk," *BYTE*, Aug 1988.
15. Ladden, R.M., "A Survey of Issues to be Considered in the Development of an Object-Oriented Development Methodology for Ada," *Softw. Eng. Notes*, Jul 1988.
16. Masiero, P.C., and F.S.R. Germano, "JSD as an Object Oriented Design Method," *Softw. Eng. Notes*, Jul 1988.
17. Pascoe, G.A., "Elements of Object-Oriented Programming," *BYTE*, Aug 1988.
18. Seidewitz, E.V. and M. Stark, *General Object-Oriented Development*, unpublished NASA GSFC report, NASA-TM-89375, Aug 1986.
19. Seidewitz, E.V. and M. Stark, "Towards a General Object-Oriented Software Development Methodology", *Ada Letters*, Vol 7, No 4, Jul-Aug 1987.
20. Thomson-Burtek, Inc., *Ada Simulation Validation Program Final Report*, with appendices, 1988.
21. Yourdon, E., and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice-Hall, 1979.
22. Ada Simulator Validation Program, Lessons Learned Seminar, Mar 1, 1988.



RUSSELL J. BROWN

Russ Brown is a software systems engineer with the MR Applications Group at GE Medical Systems, Waukesha, WI. From 1985 to 1988, Mr. Brown was an officer in the U.S. Air Force working for the Training Systems Systems Program Office at Wright-Patterson AFB, Dayton, OH. He has a B.S.E.E from the University of Wisconsin - Madison, and an M.S. in Computer Science from Wright State University in Dayton.



VERLYNDA DOBBS

Verlynda Dobbs received her Ph.D. in computer science from The Ohio State University in 1985. Her research interests are in the areas of software engineering, artificial intelligence, and Ada for artificial intelligence. Dr. Dobbs is currently on the faculty of the Department of Computer Science and Engineering at Wright State University, Dayton, Ohio 45435.

AUTHORS INDEX

Name	Page	Name	Page
Agrawal, J. C.	295	Guindi, D. S.	463
Amoroso, E.	266	Gunderson, E. P.	44
Angel, M.	122	Hager, J. A.	475
Aragon, R. W.	537	Harrison, G. C.	404
Arden, W.	114	Hartman, S.	218
Arico, F.	443	Hay, R. W.	245
Bagley, D.	251	Healer, G.	383
Balley, S. A.	13	Hetzron, J.	557
Barkataki, S.	362	Jazaa, A. T.	469
Barlev, S.	557	Johnson, M.	224
Bazzi, N.	519	Jones, A. M.	456
Bender, M.	139, 154	Jones, D. W.	528
Biau Liu, D.	576	Juozitis, P.	122
Bostic, G. E., II	44	Kelly, J.	362
Bozeman, R.	456	Kolofske, B.	342
Brashear, P.	522	Laird, J. D.	13
Brereton, O. P.	469	Land, K.	251
Brown, R.	589	Latour, L.	434
Buchman, C. D.	549	Leach, R. J.	109, 270
Burgermeister, L.	494	Lee, A. J.	130
Byrnes, C.	511	Lee, P. N.	278
Carlson, G.	209	Lefkowitz, S.	139
Carruthers, C. L.	584	Levitz, M.	557
Carter, B. J., Jr.	584	Loftus, W. P.	326
Carter, J. R.	342	Mackey, S. R.	9
Casado, B.	519	Macre, W. R.	132
Cavally, P.	87	Margono, J.	239
Chan, K. V.	193	Marino, C.	87
Chen, T. L.	411	Mayes, L.	537
Chung, A. C.	257	McCracken, W. M.	463
Clarson, D.	67	McCullough, S. J.	93
Coe, D.	25	Minder, K.	213
Cogan, K. J.	87	Moinlan, F.	313
Coleman, D.	109	Mollard, L. D.	494
Cook, P. P.	37	Montgomery, D. G.	262
Cupak, J. J., Jr.	483	Moore, F. L.	316
Dale, T.	30	Mull, A. J.	172
Davanzo, P.	557	Muralidharan, S.	188
Dobbs, V. S.	37, 589	Najjar, M. M.	197
Doberkat, E. E.	390	Nguyen, T. D.	266
Doi, D. K.	132	Oel, C. L.	326
Ellison, K. S.	51	Peavy, C.	58
Elrad, T.	197	Petersen, C. G.	288
Fedchak, E.	368	Pirchner, R.	67
Ferguson, W. D.	584	Pottinger, D.	358
Fitzgibbon, J.	58	Preston, D.	368
Fong, T.	284	Purdy, D. P.	303
Fortin, P.	316	Quinones, R.	342
Foy, R. A.	326	Richman, M. S.	288
Frush, J. A.	226	Rivera, I.	342
Fuhr, D. C.	288	Robinson, K.	24
Gallagher, E. J., Jr.	368	Rodericks, D.	342
Gargaro, A.	443	Rudolph, R. S.	307
George, K. M.	567	Rugaber, S.	463
Ginsberg, M.	67	Schacht, E. N.	78
Goel, A.	145	Scholtz, J.	182
Gopal, R.	230	Schwartz, M. I.	245
Goulet, W. J.	51	Serkin, M. B.	100
Graham, N.	173	Shah, S. N.	162
Grasso, J. M.	494	Shastri, S.	128
Greene, H.	139	Smithmier, L., Jr.	168
Griest, T. E.	154	Sobkiw, W.	411
Grosberg, L.	25	Sodhi, J.	321, 567
Gullfoyle, R.	67	Solderitsch, J.	419

Name	Page
Staples, K. A., Jr.	584
Tamboli, A.	278
Tamburro, H.	251
Terrien, D.	537
Thalhamer, J. A.	326, 419
Thompson, G. R.	186
Trost, J.	537
Tsung-Iuang, W.	193
Tupper, K.	557
Vaughn, D. A.	44
Vega, M.	251

Name	Page
Von Gerichten, L.	67
Walker, J. E.	239
Wallnau, K. C.	419
Washington, R.	224
Wheeler, T. J.	333
Wiedenbeck, S.	182
Williams, P.	522
Wilson, S.	522
Wise, C.	584
Ziegler, J.	494